KernJC: Automated Vulnerable Environment Generation for Linux Kernel Vulnerabilities

Bonan Ruan
National University of Singapore
Singapore
r-bonan@comp.nus.edu.sg

Jiahao Liu*
National University of Singapore
Singapore
jiahao99@comp.nus.edu.sg

Chuqi Zhang
National University of Singapore
Singapore
chuqiz@comp.nus.edu.sg

Zhenkai Liang
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

Linux kernel vulnerability reproduction is a critical task in system security. To reproduce a kernel vulnerability, the vulnerable environment and the Proof of Concept (PoC) program are needed. Most existing research focuses on the generation of PoC, while the construction of environment is overlooked. However, establishing an effective vulnerable environment to trigger a vulnerability is challenging. Firstly, it is hard to guarantee that the selected kernel version for reproduction is vulnerable, as the vulnerability version claims in online databases can occasionally be incorrect. Secondly, many vulnerabilities cannot be reproduced in kernels built with default configurations. Intricate non-default kernel configurations must be set to include and trigger a kernel vulnerability, but less information is available on how to recognize these configurations.

To solve these challenges, we propose a patch-based approach to identify real vulnerable kernel versions and a graph-based approach to identify necessary configs for activating a specific vulnerability. We implement these approaches in a tool, KernJC, automating the generation of vulnerable environments for kernel vulnerabilities. To evaluate the efficacy of KernJC, we build a dataset containing 66 representative real-world vulnerabilities with PoCs from kernel vulnerability research in the past five years. The evaluation shows that KernJC builds vulnerable environments for all these vulnerabilities, 32 (48.5%) of which require non-default configs, and 4 have incorrect version claims in the National Vulnerability Database (NVD). Furthermore, we conduct large-scale spurious version detection on kernel vulnerabilities and identify 128 vulnerabilities that have spurious version claims in NVD. To foster future research, we release KernJC with the dataset in the community.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**; **Software security engineering**.

*Corresponding author

RAID 2024, September 30–October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678891

KEYWORDS

Vulnerable Environment; Reproduction; Linux Kernel

ACM Reference Format:
Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. 2024. KernJC: Automated Vulnerable Environment Generation for Linux Kernel Vulnerabilities. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3678890.3678891

1 INTRODUCTION

The Linux system has become one of the cornerstones of our modern computing infrastructure, supporting a wide range of devices and services, such as cloud servers and containers, Android devices, and IoT nodes. Breaches of Linux systems can precipitate catastrophic consequences.

A primary attack surface within the Linux ecosystem is the Linux kernel, or the *kernel*. It is the ultimate line of defense and the gatekeeper of Linux system security, where the exploitation of kernel vulnerabilities could result in various severe impacts, such as privilege escalation and denial of services on traditional servers, rooted Android devices [80], and container escaping in cloud-native environments [81]. As a substantial and complex project, the Linux kernel comprises over 28 million lines of code [43]. Kernel vulnerabilities can have far-reaching impacts, while their emergence seems endless. As shown in Figure 1, the annual number of reported vulnerabilities in the upstream kernel has been trending upwards, with a notable increase in high or critical severity vulnerabilities [54], which accounts for more than 40% in 2023. This trend underscores an escalating risk profile, necessitating sustained and focused security measures to safeguard systems against exploitation.

Kernel vulnerability reproduction and replay are often an essential part of assessing the vulnerability's severity and impact, designing solutions and mitigation techniques, and evaluating the effectiveness of solutions. Also, the reproducibility of vulnerabilities emerges as a crucial factor in their prioritization. Prior studies indicate that vulnerabilities lacking reproducibility are often overlooked [57, 71], thereby leaving systems exposed to potential threats. Furthermore, by reproducing kernel vulnerabilities, analysts can comprehend the attack behaviors at runtime to update the intrusion detection systems, which plays a pivotal role in detecting and preventing future vulnerability exploitation attacks.

To reproduce a kernel vulnerability, there are two crucial elements: the vulnerable environment, and the Proof of Concept (PoC) program. The role of the vulnerable environment is to guarantee the existence and accessibility of the vulnerability in question, thereby establishing a suitable setting for analysis. On the other hand, the PoC is specialized for triggering this vulnerability. By executing the PoC in the environment, security analysts complete the reproduction and then gain further intelligence for the vulnerability.

The majority of existing research tends to concentrate primarily on the intricacies and development of PoC [14–16, 33, 51, 72, 77, 78, 83]. However, constructing the vulnerable environment is also an important but overlooked direction. Several solutions [15, 16, 51, 72] are concentrated on the static and dynamic identification of critical kernel objects to bypass specific mitigation mechanisms and facilitate kernel vulnerability exploitation. You *et al.* [83] focuses on the automated generation of PoC for kernel vulnerabilities. [14, 33, 50, 77, 78] concentrate on the automated generation and migration of ExPloit (ExP) for certain kernel vulnerabilities.

However, constructing an appropriate reproduction environment for a given Common Vulnerabilities and Exposures (CVE) ID often presents considerable challenges [13, 57]. In particular, there are two primary challenges. First, it is challenging to guarantee that the selected kernel version for reproduction is vulnerable, in that the claimed vulnerable versions in online databases, such as the National Vulnerability Database (NVD) [64], may occasionally be erroneous [5], leading to tremendous efforts wasted on building and testing non-vulnerable kernels. For example, the NVD's claim regarding kernels up to v5.12 being vulnerable to CVE-2021-22555 [63] is inaccurate; kernels from v5.11.15 to v5.11.22, within the alleged range, have already been patched. The real latest vulnerable version is v5.11.14. Second, lots of vulnerabilities cannot be reproduced in kernels built from default upstream configurations (referred to hereafter as *configs*). it is time-consuming to explore the configs to identify and enable the necessary kernel configs responsible for the activation of a specific vulnerability, due to the complexity of the kernel configuration system.

Vulnerabilities often occur in specific subsystems or as a result of particular module features, which may not be active by default. Nevertheless, this important detail is frequently omitted in both CVE databases and external reports, complicating the process of activating kernel vulnerabilities. What's worse, the reliability of config files from successful reproduction attempts is also questionable, as evidenced by sources like [10, 11] and our observations of vulnerability disclosure [3, 42] in § 5.5.1. Similar to the situation in [13], it is not practical to enable all kernel configs as well, as some configs are mutually exclusive, and others may affect the execution of target functionality. Although there is an `allyesconfig` option available in the Linux kernel build mechanism enabling as many configs as possible, our manual experiments and prior work [24, 83] confirm that the kernel built with `allyesconfig` is not bootable.

In this paper, we introduce KernJC[1], a novel tool that adopts a patch-based approach for pinpointing inaccuracies in version claims of online databases and accurately determining the actual vulnerable kernel version. Additionally, KernJC utilizes a graph-based approach to autonomously ascertain the specific kernel config set required for each vulnerability. KernJC also provides an intuitive command-line interface for managing kernel vulnerabilities and deploying PoCs.

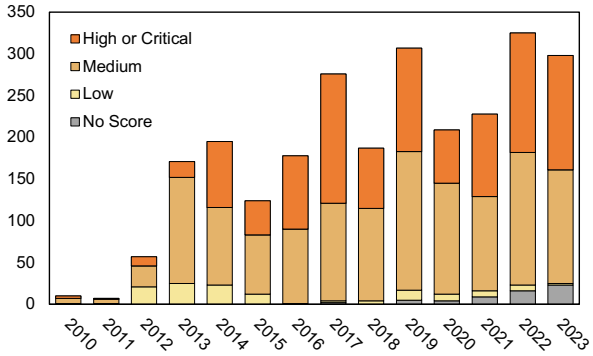

Figure 1: Numbers of Kernel Vulnerabilities since 2010

Our implementation of KernJC, when applied to analyze kernel vulnerabilities, identifies 128 instances of false positive version claims in the NVD database. Furthermore, we test KernJC against 66 kernel vulnerabilities associated with PoCs in prominent research publications [1, 8, 14–16, 31–33, 44–47, 52, 68, 72, 75, 77, 82, 84–86, 89] from main security conferences over the past five years. The evaluation results reveal KernJC's efficacy in accurately establishing vulnerable environments and accomplishing the reproduction for all these vulnerabilities, with 48.5% requiring non-default kernel configs. Notably, KernJC also uncovered 4 out of the 66 vulnerabilities that have incorrect version claims in the NVD database.

To the best of our knowledge, this is the first work focusing on the automated generation of the reproduction environment for Linux kernel vulnerabilities. In particular, our work makes the following contributions:

- We develop techniques to correct and enhance the information available for kernel CVEs, fixing version errors and adding essential configs needed by reproduction. Especially, our research finds 128 cases of incorrect vulnerability version claims in the NVD database, emphasizing the need to improve vulnerability reporting accuracy.
- We implement our solution as an open-source tool, KernJC[2], with a user manual and a demonstration example [3] in this repository for facilitating the usage.
- We evaluated KernJC with representative real-world kernel vulnerabilities. The evaluation showcases KernJC's efficacy in constructing and automating the setup of vulnerable environments, where the target vulnerabilities are available and accessible from userland.
- We have compiled and openly shared a dataset [4] comprising 2,256 kernel vulnerabilities, 1,829 of which have verified vulnerable versions (whose version ranges are claimed in NVD database), 1,633 of which have identified kernel configs (exclusive of vulnerabilities that do not rely on any kernel

[1] KernJC stands for Kernel JiaoChang, where JiaoChang, in ancient China, referred to a site dedicated to military training and competition.

[2] KernJC: https://github.com/NUS-Curiosity/KernJC

[3] Demonstration example: https://github.com/NUS-Curiosity/KernJC/blob/main/README.md#quick-start

[4] Dataset: https://github.com/NUS-Curiosity/KernJC/tree/master/db

config), and 66 of which are paired with functional PoCs. This dataset is currently the most extensive collection of kernel vulnerabilities and their PoCs known to us.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the process of kernel vulnerability reproduction, emphasizing the core focus of our study — constructing environments susceptible to kernel vulnerabilities. Following this, we use `CVE-2021-22555` as a concrete example to illustrate the challenges encountered in creating such vulnerable environments.

2.1 Kernel Vulnerability Reproduction

Once a kernel vulnerability is disclosed, the standard reproduction process by analysts involves two key stages: constructing the corresponding vulnerable kernel environment and developing a workable Proof of Concept (PoC) program.

The goal of constructing vulnerable environments is to create an environment with a specific vulnerability accessible to users for interaction or testing. This process requires the analyst to identify a kernel version known to be vulnerable, enable necessary configs, and compile the kernel image. Subsequently, the analyst has two options: either install the kernel on a physical machine and reboot to activate the vulnerable version or utilize virtualization tools like QEMU [7] to set up a virtual machine tailored for the reproduction task. In the virtualization scenario, integrating a corresponding root file system with the kernel image is crucial for a fully operational environment. A prevalent method for constructing the root file system is using the script from the syzkaller project [23], a well-known unsupervised coverage-guided kernel fuzzer [21].

PoC development involves crafting a functional PoC using available vulnerability descriptions, patches, and technical reports. More specifically, a PoC is a custom program created to attack the kernel-vulnerable environment, often leading to kernel crashes or inducing a certain kernel state (*e.g.,* terminated), thereby demonstrating the reproducibility of a vulnerability.

Successfully reproducing a kernel vulnerability is the cornerstone of assessing its severity [57]. Recent efforts have primarily concentrated on the development of PoC exploits to trigger these vulnerabilities [14, 33, 50, 77, 78, 83]. For instance, SemFuzz [83] utilizes various vulnerability-related data, including descriptions, source code, and patches, to generate PoCs for kernel vulnerabilities autonomously. However, the construction of a vulnerable kernel environment often receives insufficient attention, complicating the process of vulnerability reproduction. To address this gap, this study is dedicated to *identifying challenges and automating the generation of vulnerable kernel environments*. We aim to simplify the process of reproducing kernel vulnerabilities, further reducing the time required to evaluate the risk and collect runtime attack behaviors associated with various kernel vulnerabilities.

2.2 Challenges

Motivating Example. We discuss the challenges faced by kernel-vulnerable environment construction with an example of the real-world kernel vulnerability, `CVE-2021-22555` [63], an out-of-bounds issue in the Netfilter subsystem with the potential for local privilege escalation.

NVD's vulnerability record for CVE-2021-22555 delineates the affected kernel versions as those up to, but not including, v5.12 [63]. After cross-checking with the official Linux kernel release list [49], it appears that versions up to and including v5.11.22 are vulnerable to CVE-2021-22555, in line with NVD's assertion, implying that we can select any version from this range for vulnerability reproduction.

To reproduce this kernel vulnerability, we first compile the v5.11.22 kernel to create the vulnerable environment, then we test it with the public PoC [20], and it fails. This failure is attributed to the presence of the patch for CVE-2021-22555 in the v5.11.22 source code. Specifically, the three red lines from the vulnerable function `xt_compat_target_from_user` in Figure 2 (a), which are deleted by the patch [76] in Figure 2 (b), have already disappeared in the v5.11.22 source code [25]. Moreover, our investigation confirms that kernel versions from v5.11.15 to v5.11.22 have similarly been patched against this vulnerability as well. Consequently, the correct upper boundary for vulnerable kernels is v5.11.14 (inclusive). Reproduction tests conducted on the v5.11.14 kernel using the public PoC confirm its susceptibility to CVE-2021-22555. Therefore, attempting reproduction with v5.11.22 kernels would lead to an ineffective allocation of resources and time.

Following the determination of the vulnerable kernel version, the subsequent phase involves enabling the appropriate kernel configs to ensure the vulnerability is available and accessible in target kernel before initiating the kernel build. In terms of CVE-2021-22555, our analysis of a segment of its patch [76], as illustrated in Figure 2 (b), identifies crucial configs in the Makefiles along the path leading to the inclusion of `net/netfilter/x_tables.c`. These include `CONFIG_NETFILTER` and `CONFIG_NETFILTER_XTABLES` in Figure 2 (c), as well as two other configs, `CONFIG_NET` and `CONFIG_INET`, upon which the former two configs depend, as specified in the Kconfig files in Figure 2 (d). Furthermore, within the vulnerable file itself in Figure 2 (a), a conditional compilation directive (`#ifdef`) is observed, controlling the inclusion of the vulnerable function based on `CONFIG_COMPAT`. By conducting a thorough manual analysis of all files affected by the patch in this manner, we can compile a comprehensive list of configs **necessary** to ensure the presence of the vulnerability in the target kernel version. The detailed analysis results for CVE-2021-22555, including the essential configs, are presented in Figure 2 (e).

Unfortunately, even with the heuristic analysis above, the PoC for CVE-2021-22555 [20] remains non-functional. The issue stems from the PoC's specification of `NFQUEUE` as the target in Netfilter, depicted in a code snippet from the PoC in Figure 2 (f). This additionally requires the activation of `CONFIG_NETFILTER_XT_TARGET_NFQUEUE` to support the specified target. Ultimately, enabling this config allows the PoC to trigger the vulnerability successfully.

We identify two challenges that arise from the reproduction process of CVE-2021-22555:

C1: Inaccurate Vulnerability Versions in CVE databases. The version range documented in vulnerability databases is not always accurate, requiring cross-checking with the kernel release list and the source code repository to identify a vulnerable version, which is quite time-consuming.

C2: Non-obvious Vulnerability Configs. Vulnerability descriptions and reports often lack detailed information regarding the

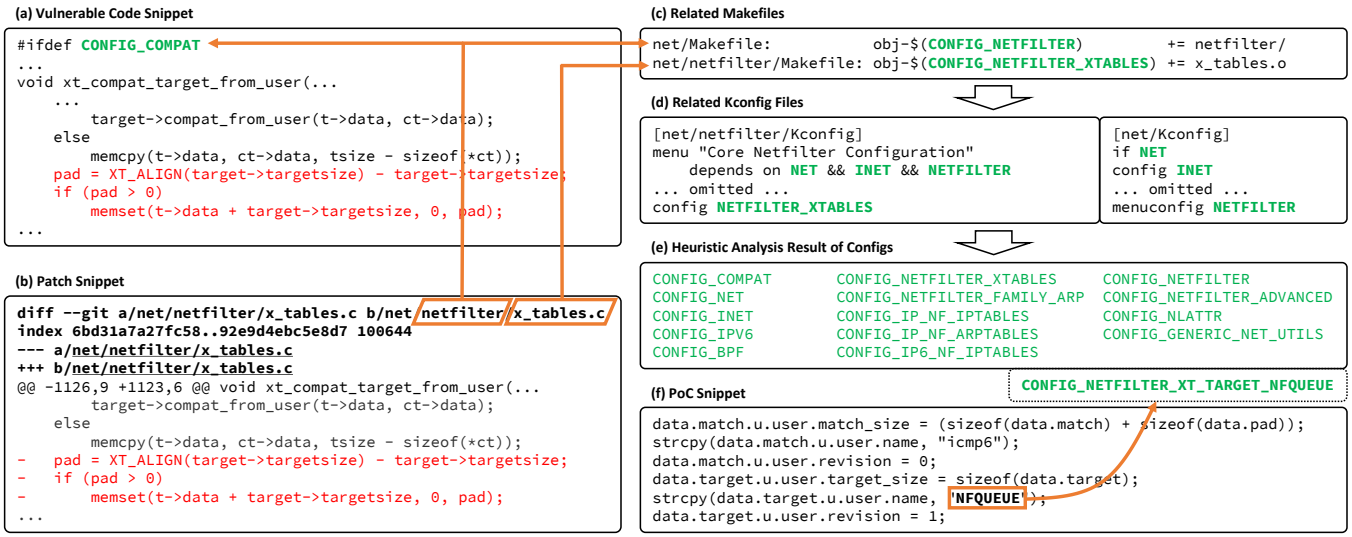

Figure 2: An Example Vulnerability from CVE-2021-22555. (a) shows one snippet from the source code of this vulnerability in the v5.11.14 kernel. (b) shows one snippet of the patch for the vulnerability. (c) shows the Makefiles related to the source code snippet from `net/netfilter/x_tables.c`. (d) shows the Kconfig files related to `net/netfilter/x_tables.c`. (e) shows the heuristic config analysis result based on the paths and contents of vulnerable files, patch, and the Kconfig&Kbuild mechanisms. (f) shows one snippet from the public PoC [20] for this vulnerability. The orange boxes and arrows illustrate the derivation of configs.

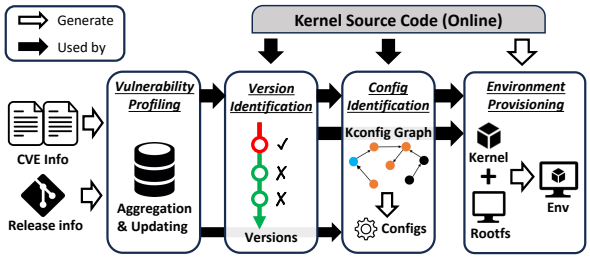

Figure 3: Overview of KernJC Architecture

necessary configs that would ensure both the presence and user space accessibility of the vulnerability. Analysts have to download the source code of a vulnerable version, then compile the kernel using default configs (`make defconfig`) and execute the PoC program to ascertain if it either crashes the system or triggers a Kernel Address SANitizer (KASAN) report.

This approach is generally effective for vulnerabilities located in the kernel's core functionalities, which are usually enabled by default configs. However, for numerous vulnerabilities tied to non-default functionalities or intricate subsystems, such as CVE-2021-22555, relying solely on default configs proves inadequate for their activation. A manual analytical approach is typically needed to identify additional required configs. Alternatively, a heuristic analysis of the vulnerable code is often used alongside an examination of the Kbuild [38] and Kconfig [37] mechanisms. Despite its monolithic nature, the kernel source code is hierarchically organized and developed modularly. The Kconfig and Kbuild systems work in tandem to tailor the kernel's functionalities and facilitate its build process. Statistically, the latest Linux kernel encompasses over 15,000 config items (`CONFIG_*`), the majority of which are instrumental in deciding whether to incorporate specific source code files and activate particular features. Typically, kernel vulnerabilities manifest at the code block level. Thus, pinpointing the essential configs for introducing a vulnerability entails selecting the relevant configs from these 15,000 items, ensuring the inclusion of files and code blocks associated with the vulnerability. Given the extensive number of items and their intricate interrelations, manually undertaking this task is notably labor-intensive. What's worse, some configs needed for the reproduction are determined by analyzing the PoC programs, which are not always readily available, particularly for vulnerabilities that have only recently been disclosed.

3 APPROACH

3.1 Overview

Figure 3 delineates the architectural framework of KernJC, which is segmented into four distinct components: (1) a **vulnerability profiling** component, dedicated to the continuous aggregation and progressive updating of data concerning kernel vulnerabilities; (2) a **vulnerability version identification** component, tasked with pinpointing an actual vulnerable kernel version corresponding to a specific CVE ID; (3) a **vulnerability config identification** component, focused on determining the requisite kernel configs for activating a particular vulnerability; and (4) an **environment provisioning** component, instrumental in constructing the target kernel and establishing the comprehensive environment necessary for reproducing the vulnerability.

Vulnerability Profiling. KernJC is executed routinely to gather kernel vulnerability information for profiling each vulnerability. The vulnerability profiles encompass essential information about vulnerabilities, patches, and kernel versions sourced from online

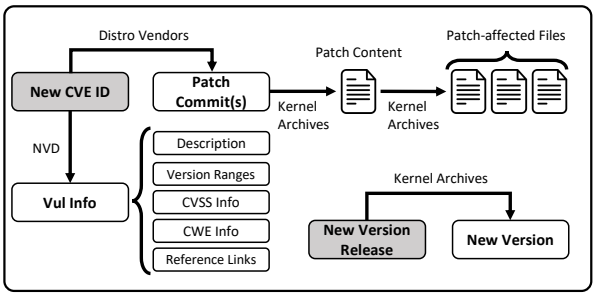

Figure 4: Vulnerability Profiling

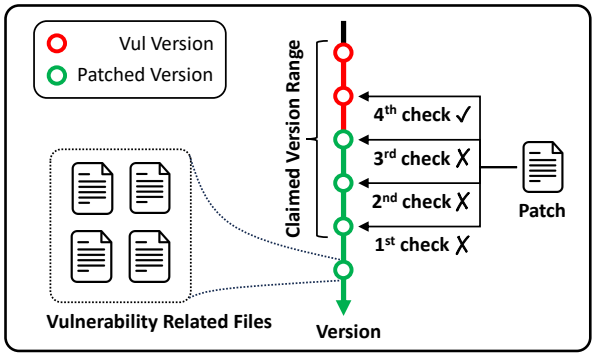

Figure 5: Vulnerability Version Identification

CVE databases and official kernel repositories for subsequent analysis. We will detail the vulnerability profiling component in § 3.2.

Vulnerability Version Identification. Upon receiving a CVE ID, KernJC validates the claimed version range's accuracy and identifies an authentically vulnerable version. To do so, KernJC ascertains the alleged vulnerable version ranges, correlated patches, and the implicated source code files to scrutinize the presence of patches within the purportedly vulnerable versions. KernJC designates a version as a false positive if a patch is detected within it. Subsequently, a retrospective examination is conducted through the kernel release chronology until a version devoid of the patch is located. The vulnerability version identification component is delineated in § 3.3.

Vulneability Config Identification. Once KernJC identifies the actual vulnerable kernel version, it proceeds to identify the specific kernel configs to activate the vulnerability. To streamline this intricate analysis, we propose a graph-based approach. The fundamental principle is to derive intuitive *direct* configs from the vulnerability profiling component, build a Kconfig graph for the target kernel source code, and identify *hidden* configs holding special relations with the *direct* configs in the graph. The *direct* and *hidden* configs serve as the ultimate identification result. The vulnerability config identification component is detailed in § 3.4.

Environment Provisioning. After KernJC discerns the kernel configs essential for the vulnerability, it integrates these configs with the foundational ones (`defconfig`). Following this integration, KernJC proceeds to compile the kernel's source code of the identified vulnerable version, to build the kernel image. This constructed image, in conjunction with a root filesystem (rootfs), is then utilized to provision a virtual machine as the final reproduction environment atop hypervisors.

3.2 Vulnerability Profiling

Figure 4 illustrates our methodology for the incremental profiling of kernel vulnerabilities. Drawing from our hands-on experience in reproducing vulnerabilities, we identify three pivotal categories of important information:

Vulnerability Information. Given a CVE ID, the preliminary step involves comprehending its scope through its description. In rare cases, the description also includes the kernel configs needed to trigger the vulnerability [60]. The stated range of vulnerable versions aids in targeting a specific kernel version, although it can occasionally be inaccurate, as exemplified in § 2.2. Metrics such as CVSS and Common Weakness Enumeration (CWE) offer an initial severity assessment. Additionally, references and reports provide analyses from other experts.

Patch Information. Patches are invaluable in offering spatial and temporal insights into vulnerabilities. They assist in pinpointing the affected files and functions, understanding the root cause, and deriving strategies for triggering and exploitation. Patches are also instrumental in verifying the vulnerability of a targeted version and deriving the necessary kernel configs, providing pivotal information for reproduction.

Release Version Information. Maintaining an updated list of release versions streamlines the selection of an appropriate version for vulnerability reproduction. The version list is mainly used to fill the vulnerable version range used in § 3.3.

When a new CVE of Linux kernel is reported, KernJC's initial step is to gather CVE details from NVD. Subsequently, it obtains patch commit IDs from distribution vendors (such as Ubuntu, Red Hat, and SUSE) for in-depth analysis. It is noteworthy that while NVD provides patch information for certain kernel vulnerabilities, the extent of this data is often surpassed by that from the vendors. Moreover, the occasional mislabeling of bug introduction commits as 'Patch' in NVD [63] complicates the identification of the actual patch link, leading us to prioritize vendor sources over NVD for patch information. Concurrently, our approach involves periodically reviewing new kernel releases, and updating the local list with any newly released versions.

3.3 Vulnerability Version Identification

To detect spurious version range claims and select a real vulnerable kernel version, we propose a patch-based approach, shown in Figure 5. At a high level, KernJC inspects whether the corresponding patch to the vulnerability has been applied, by checking kernel versions in chronological order.

KernJC initially maps the claimed range of vulnerable versions with the list of kernel version releases. It then proceeds to verify the presence of the patch in each version, starting from the upper boundary of the inclusive version range and moving downwards. This process continues until it reaches the first version where the patch is not found. Specifically, KernJC attempts to apply the patch to each kernel version. If the version is already patched, a "re-patch" will be detected when applying the same patch again. By doing so, KernJC identifies the first version where the patch is not found

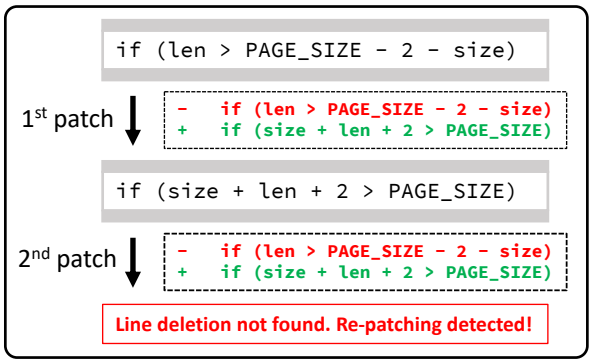

Figure 6: Detection of Re-patching for CVE-2022-0185. The grey rectangles around the target line represent the context.

(i.e., the version without re-patching happens when applying the patch). The version without a patch likely indicates the existence of the specific vulnerability.

Figure 6 employs the patch [29] for kernel vulnerability CVE-2022-0185 [62] as an example to elucidate the process of re-patching detection. This specific patch incorporates both a line deletion and a line addition, framed within a 3-line context (represented by the grey rectangles around the target line in Figure 6). The first patching action substitutes the original `if` statement with a new one, thereby remedying the vulnerability. Consequently, a subsequent patch attempt is unsuccessful, as the target line for deletion is no longer present following context alignment. This indicates that the source code has already undergone patching.

Essentially, the patch commit(s) content is derived from the `diff` output for affected files in kernel source code of old & new versions [40]. The finest granularity of patches is at the line level, with differences being delineated on a per-line basis, *e.g.,* line additions or line deletions. The patch for a Linux kernel vulnerability could be composed of one or more commits; each commit may contain `diff` results for one or more files; each file's `diff` result contains either line additions, line deletions, or both. However, only the record of line modifications is inadequate for patching. Location information should also be provided to locate the accurate position to be patched. Due to the activity of Linux kernel development, locating via line numbers is prone to be erroneous when other commits occur in the same file. Linux kernel community adopts an N-line context way [35] to provide location information, i.e., attaching N lines of source code before and after the modified lines as context into the patch, which can tolerate the aforementioned line shift phenomenon to some extent. Although it would still fail in extreme cases, (*e.g.,* when both the N-line context and the affected line(s) are replicated in the same file), this technique is easy to use and has been adopted in Linux kernel community for a long time.

3.4 Vulnerability Config Identification

The vulnerability config identification process is composed of three stages: direct config identification, Kconfig graph construction, and hidden config identification, as shown by the pseudo-code in Algorithm 1 (the complete algorithm is in Appendix A) and Figure 7.

Firstly, KernJC gathers the direct configs (D), composed of the description-level configs (DDC), the path-level configs (DPC), and

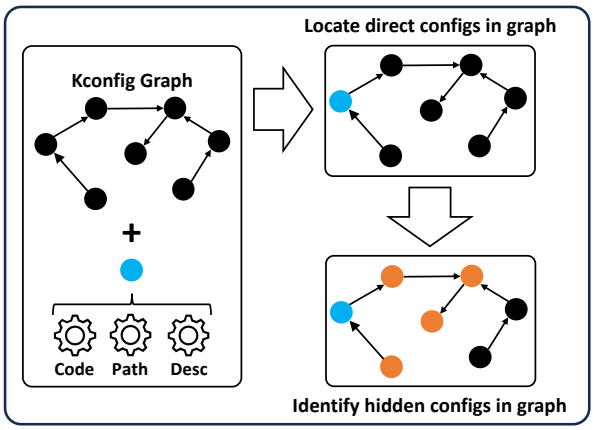

Figure 7: Vulnerability Config Identification. Black dots represent all the configs in the Kconfig graph. Blue dots represent the set of *direct configs*. Orange dots represent the set of *hidden configs*. The three gears individually represent the set of code-level, path-level, and description-level configs.

the code-level configs (DCC), by analyzing the vulnerability description, patch, vulnerable source code, and Makefile(s). DDC is derived from the description text, if any. DPC represents the set of configs responsible for including the directory path towards the vulnerable file(s). DCC represents the set of configs responsible for including the vulnerable code. D serves as the starting point for the graph analysis later on.

Secondly, KernJC starts from the root Kconfig file in the kernel source code, and recursively parses and imports new Kconfig files in sub-directories till leaf directories, to build the Kconfig graph (G). The specification of this directed graph is presented in Table 1. There are three types of config definition in the Kconfig mechanism: `config`, `menuconfig`, and inner `config` within blocks (*e.g.,* `choice` block), which are represented by vertexes in our graph. Additionally, `menu` is a special item in Kconfig, which also has relations with other configs, but it is not a real config. KernJC takes it as a virtual vertex in the graph to ensure the connectivity and filters it out after getting all the necessary configs. Besides vertexes, four types of directed edges are defined in the graph, which are derived from the specification of the Kconfig mechanism. For example, as shown in § 2.2, the `menu "Core Netfilter Configurations"` *depends on* the following three configs: `CONFIG_NET`, `CONFIG_INET`, and `CONFIG_NETFILTER`. As a result, there will be three depend edges from the virtual vertex `"Core Netfilter Configurations"` to the three config vertexes in the graph. Similarly, `CONFIG_INET` has an opaque_depend edge to `CONFIG_NET`, as the former config is defined in the `if` block of the latter one. The `select` and `imply` edges represent the relations in Kconfig that when config A is enabled, it will `select` or `imply` config B to be enabled as well, if there is a `select CONFIG_B` statement within the definition of `CONFIG_A`.

Thirdly, KernJC leverages the direct configs and the Kconfig graph to get the hidden configs (H), *i.e.,* configs that are necessary but not directly associated with vulnerabilities, which include three types of configs (vertexes in the graph): (1) configs that are recursively reachable **from** any direct config (HRC), (2) configs holding N-hop (N is a parameter to control the relation scale)

Table 1: Kconfig Graph Specification

Element	Type	Description
config	vertex	"config" <symbol> in Kconfig
menuconfig	vertex	"menuconfig" <symbol> in Kconfig
inner_config	vertex	configs embedded in other blocks in Kconfig
menu	virtual vertex	"menu" in Kconfig
depend	edge	"depends on <expr>" in Kconfig
opaque_depend	edge	statements like "if" and "source" in Kconfig
select	edge	"select <symbol>" in Kconfig
imply	edge	"imply <symbol>" in Kconfig

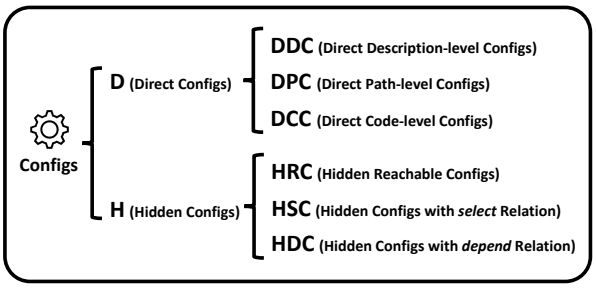

Figure 8: Config Hierarchy and Categories

`select` relation **to** any direct config (*HSC*), and (3) configs holding *N*-hop `depend` relation **to** any direct config (*HDC*). As edges in the Kconfig graph are directed, the reachability of *HRC* serves as a strong attribute to identify the vulnerability configs, which is consequently handled recursively. Nonetheless, *HSC* and *HDC* hold inverse relations to direct configs, which are only used to fulfill the config requirement for functionalities in complex subsystems. Consequently, the *N*-hop setting should be as small as possible to avoid introducing too many weakly related configs. Experiments in § 5.2 indicate that $N = 1$ works for all the tested vulnerabilities. Lastly, the set *S*, composed of both direct configs and hidden configs, serves as the final result for the vulnerability config identification process. It is noticeable that virtual vertexes are not considered as one hop, as they do not represent real configs, and will be filtered out from *S*. In summary, the identified configs can be categorized into six categories, as shown in Figure 8.

For CVE-2021-22555, the heuristic analysis result in Figure 2 (e) is equal to the combination of *DPC*, *DCC* and *HRC* derived by the graph-based approach. Especially, the last pivotal config to activate this vulnerability, `CONFIG_NETFILTER_XT_TARGET_NFQUEUE`, is successfully captured in *HDC*. Therefore, this approach can provide adequate configs to support the reproduction of CVE-2021-22555, illustrated by the result graph shown in Figure 9. Furthermore, § 5.3 will analyze the contributions of configs in different sets (*DDC*, *DPC*, *DCC*, *HRC*, *HSC* and *HDC*) to the reproduction of representative kernel vulnerabilities.

The approach to identifying necessary kernel configs for a specific vulnerability is derived from two observations (exemplified by the motivating example in Figure 2): (1) to include a vulnerability, the vulnerable code must be activated by conditional compilation like `#ifdef` (code-level configs), and the directory path towards the vulnerable file(s) must be included in Makefile(s) along the path (path-level configs); (2) the code-level and path-level configs have multiple necessary relations with other configs in the hierarchical

Algorithm 1: Vulnerability Config Identification

Input: *V*: Vulnerability Description; *P*: Patch Text for *V*; *SC*: Vulnerable Kernel Source Code
Output: *S*: Set of Identified Config

Procedure GetVulConfigs(V, P, SC)
  D = GetDirectConfigs(V, P, SC)
  G = BuildKconfigGraph(SC)
  H = GetHiddenConfigs(D, G)
  $S = D \cup H$
  return S
Procedure GetDirectConfigs(V, P, SC)
  DF = affected files mentioned in V (optional)
  DDC = configs mentioned in V (optional)
  DPC = configs in SC to enable DF and files in P
  DCC = configs in SC to enable code in P by #ifdef
  return $DDC \cup DPC \cup DCC$
Procedure BuildKconfigGraph(SC)
  G = empty graph
  RK = root Kconfig file in SC
  AddKconfigNodes(G, RK, SC)
  AddKconfigEdges(G, RK, SC)
  return G
Procedure GetHiddenConfigs(D, G)
  $H = \emptyset$
  foreach *c in D* **do**
    HRC = reachable configs from c in G
    HSC = configs with *select* relation to c in G
    HDC = configs with *depend* relation to c in G
    Add HRC, HSC, HDC into H
  end
  return H

Kconfig mechanism. Respectively, we regard configs in (1) as *direct configs*, in that they are intuitive and can be found out by analyzing the patch and source code, and the configs in (2) as *hidden configs*, because they hide in the complex structure of Kconfig and have to be identified with deep analysis. As a result, our initial idea to identify necessary configs is to first figure out all code-level (*DCC*) and path-level (*DPC*) configs, build a directed graph for all kernel configs and their relations from Kconfig files in the vulnerable source code, and use these configs as starting points to discover all reachable configs as hidden configs (*HRC*).

However, only configs above do not necessarily ensure the activation of one vulnerability. There are exceptions in the reproduction experiences. For example, to trigger CVE-2017-18344 [60], the PoC must access a pseudo-file, `/proc/pid/timers`, which is only available when `CONFIG_CHECKPOINT_RESTORE` is set [41]. For this case, the CVE description in NVD rarely provides the needed config items. Besides, for vulnerabilities located in complicated subsystems (*e.g.,* Netfilter), special configs within the subsystems are needed to ensure the normal functionality in user space, illustrated by the CVE-2021-22555 example in § 2.2. Consequently, *DDC*, *HSC*, and

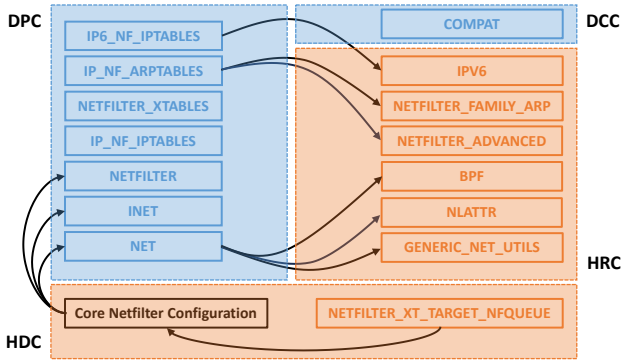

Figure 9: Graph-based Analysis for CVE-2021-22555. Blue boxes represent *direct configs*, orange boxes represent *hidden configs*, and the black box represents a menu item, which is to be filtered out at last. The `CONFIG_` prefix is omitted.

HDC are proposed to improve the identification result. Considering the complexity of Linux Kconfig mechanism, we only derive *HSC* and *HDC* from **one-hop** relations, instead of two or more hops, to avoid involving too many useless configs. This strategy proves to be effective and adequate in § 5.3.

4 IMPLEMENTATION

We develop KernJC in 3.4K lines of Python code. The key technical aspects of KernJC are detailed below.

Patch Processing. Patches play a crucial role in identifying both the presence of vulnerabilities and the necessary configs. KernJC integrates crawlers for both the Ubuntu Security Tracker [70] and Red Hat Bugzilla [26], aimed at collecting patch commit IDs related to kernel vulnerabilities. When required, KernJC utilizes these IDs to fetch the raw patch content from the official kernel repository [39]. The system then parses this data to pinpoint the modified files and specific line changes. These files are subsequently retrieved for applying patches and executing a direct config search. For identifying configs, KernJC inspects each vulnerable function's end where modifications occur, tracing back to the file's beginning to detect any conditional compilation directives (`#ifdef CONFIG_*`).

Kconfig Graph Construction. The creation of a Kconfig graph is intricate due to the complex relations within the Linux kernel's Kconfig system. For example, a `source "net/packet/Kconfig"` line within an `if NET` condition implies that all configs in the `net/packet/Kconfig` file are dependent on `CONFIG_NET`. KernJC tackles this by applying a recursive strategy to form graph edges that represent these relations. Technologically, the tool utilizes NetworkX [58] to construct the graph and support traversal analysis.

Rootfs and Virtual Machine Setup. KernJC employs an adapted version of the `create-image` script from the syzkaller project [23] for generating the base image for rootfs. In constructing each vulnerable environment, KernJC uses QEMU's [36] copy-on-write overlay feature to quickly establish rootfs for the targeted virtual machine (VM). Finally, the tool engages QEMU to launch the virtual machine, facilitating the vulnerability reproduction process.

User-friendly Interface. The final phase in vulnerability reproduction is the testing of the developed PoC within the execution environment. This traditionally involves a cumbersome process where the analyst must transfer the PoC into the environment, execute it, and then monitor its behavior. Success on the initial attempt is rare, leading to a repetitive cycle of development and debugging of the PoC until it functions as intended. To streamline this often arduous task, KernJC's command-line interface has been thoughtfully designed to resemble Docker's user-friendly interface. It includes commands like `cp` for copying files between the host and the VM, `exec` for executing commands in the VM, and `attach` for interacting with the VM, all aimed at simplifying the delivery and execution of the PoC within this VM.

5 EVALUATION

In this section, we evaluate KernJC by answering the following research questions (RQs):

RQ1: How is KernJC's performance in the reproduction of kernel vulnerabilities? (§ 5.2)

RQ2: How well do the configs identified by KernJC facilitate the reproduction of kernel vulnerabilities? (§ 5.3)

RQ3: How many false positive version claims in NVD can KernJC detect for Linux kernel vulnerabilities? (§ 5.4)

All the experiments are performed on a server with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and 128 GB physical memory. The host OS is Ubuntu 22.04.3 LTS. Experiments for vulnerabilities from 2019 are conducted directly on the host. For vulnerabilities before 2019, we spawn a Ubuntu 18.04 container with Docker on this host for experiments, which avoids the incompatibility between the new compiler on the host and the old kernel source code.

5.1 Dataset

To adequately investigate the efficacy of our approaches, we create a comprehensive dataset of kernel vulnerabilities encompassing as many vulnerabilities as possible. We collect the CVE data (sourced from NVD) and the corresponding patch commits (from Linux distribution vendors, such as Ubuntu and Red Hat) for kernel vulnerabilities to date. After filtering out the invalid CVE IDs and those without patch commits recorded, we get 2,256 vulnerabilities. The specifics of this data collection are elaborated in § 3.2. The 2,256 vulnerabilities serve as the whole dataset, among which we successfully identify the vulnerable versions for 1,829 vulnerabilities, and identify config(s) for 1,633 vulnerabilities, as shown in Figure 10. It should be noted that the version identification process aims to verify the correctness of existing version ranges, which is only done for kernel vulnerabilities whose vulnerable version ranges have already been claimed in NVD database. As a result, old CVEs without explicit version ranges in NVD are excluded. Furthermore, the absence of configs for a particular vulnerability does not imply KernJC 's inability to identify the necessary configs, as specific kernel vulnerabilities' availability and accessibility are not contingent on any kernel config.

Although covering more vulnerabilities could lead to a better evaluation of KernJC, conducting reproduction for the entire dataset is infeasible, since most vulnerabilities in the wild lack workable PoCs. Moreover, it requires plenty of manual efforts to develop PoCs for more than 1.6K vulnerabilities. Instead, to comprehensively investigate KernJC's performance, we move our focus

Table 2: Vulnerability Reproduction Results. "RwKC?" and "RwDC?" denote reproducibility with KernJC-identified and default configs, respectively. "FPV?" indicates false positive version claims in NVD. Abbreviations: UAF = Use after Free; OOB = Out of Bounds; TOCTOU = Time of Check to Time of Use; DF = Double Free; ND = Null-pointer Dereference.

ID	Type	CVSS	Subsystem	RwKC?	RwDC?	FPV?	Paper(s)
CVE-2016-10150	UAF	9.8	virt/kvm	✓	✗	✗	K(H)eaps, ELOISE, SLAKE, Kepler
CVE-2016-4557	UAF	7.8	kernel/bpf	✓	✗	✗	AEM, K(H)eaps, ELOISE, SLAKE, Kepler, RetSpill
CVE-2016-6187	OOB	7.8	security/apparmor	✓	✗	✗	Pspray, PET, AEM, K(H)eaps, ELOISE, KOOBE, SLAKE, Kepler, RetSpill
CVE-2017-16995	Logic	7.8	kernel/bpf	✓	✗	✗	AEM, Kepler
CVE-2017-18344	OOB	5.5	kernel/time	✓	✗	✗	PET, AEM
CVE-2017-2636	DF	7.0	drivers/tty	✓	✗	✗	SegFuzz, PET, AEM, K(H)eaps, ExpRace, ELOISE, SLAKE, Kepler, Razzer, RetSpill
CVE-2017-6074	DF	7.8	net/dccp	✓	✗	✗	Pspray, AEM, K(H)eaps, ELOISE, SLAKE, Kepler, RetSpill
CVE-2017-8824	UAF	7.8	net/dccp	✓	✗	✗	PET, AEM, K(H)eaps, SLAKE, Kepler, RetSpill
CVE-2018-12233	OOB	7.8	fs/jfs	✓	✗	✗	ELOISE
CVE-2018-5333	ND	5.5	net/rds	✓	✗	✗	AEM
CVE-2018-6555	UAF	7.8	net/irda	✓	✗	✗	Pspray, AlphaEXP, AEM, K(H)eaps, ELOISE, SLAKE, RetSpill
CVE-2019-6974	UAF	8.1	virt/kvm	✓	✗	✗	SegFuzz, ExpRace
CVE-2020-14381	UAF	7.8	futex	✓	✓	✓	AlphaEXP
CVE-2020-16119	UAF	7.8	net/dccp	✓	✗	✗	PET, DirtyCred
CVE-2020-25656	UAF	4.1	drivers/tty	✓	✓	✓	DDRace
CVE-2020-25669	UAF	7.8	drivers/input	✓	✗	✗	StateFuzz
CVE-2020-27194	OOB	5.5	kernel/bpf	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2020-27830	ND	5.5	drivers/accessibility	✓	✗	✗	StateFuzz
CVE-2020-28941	ND	5.5	drivers/accessibility	✓	✗	✗	StateFuzz
CVE-2020-8835	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred
CVE-2021-22555	OOB	7.8	net/netfilter	✓	✗	✓	PET, AlphaEXP, DirtyCred
CVE-2021-26708	UAF	7.0	net/vmw_vsock	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2021-27365	OOB	7.8	drivers/scsi	✓	✗	✗	Hybrid, DirtyCred, RetSpill
CVE-2021-34866	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred
CVE-2021-3490	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred, RetSpill
CVE-2021-3573	UAF	6.4	net/bluetooth	✓	✗	✓	AlphaEXP
CVE-2021-42008	OOB	7.8	drivers/net	✓	✗	✗	Hybrid, AlphaEXP, DirtyCred
CVE-2021-43267	OOB	9.8	net/tipc	✓	✗	✗	Hybrid, AlphaEXP, KRover, DirtyCred, PET, RetSpill
CVE-2022-0995	OOB	7.8	watch_queue	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2022-1015	OOB	6.6	net/netfilter	✓	✗	✗	PET
CVE-2022-25636	OOB	7.8	net/netfilter	✓	✗	✗	AlphaEXP, DirtyCred, RetSpill
CVE-2022-32250	UAF	7.8	net/netfilter	✓	✗	✗	Hybrid
CVE-2022-34918	OOB	7.8	net/netfilter	✓	✗	✗	PET, Hybrid
CVE-2023-32233	UAF	7.8	net/netfilter	✓	✗	✗	Hybrid

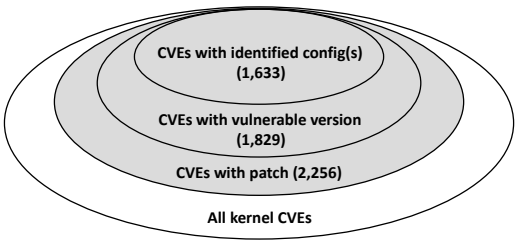

Figure 10: Construction of the Whole Dataset

to vulnerabilities that received widespread attention, which is sufficient to demonstrate KernJC's practical in the construction of vulnerable kernel environments. Consequently, we meticulously curated a subset of 66 kernel vulnerabilities in prominent research publications (SHARD [1], Midas [8], KOOBE [14], ELOISE [15], SLAKE [16], Razzer [31], SegFuzz [32], AEM [33], Pspray [44], Hybrid [46], DirtyCred [47], LinKRID [52], KRover [68], AlphaExp [72], PET [75], Kepler [77], PAL [82], DDRace [84], K(H)eaps [85], RetSpill [86], StateFuzz [89]) from ACM CCS, IEEE S&P, NDSS and USENIX Security conferences in the past five years, with their associated PoCs from the Internet. This subset spans over nine years and covers a wide array of vulnerability types in various Linux kernel subsystems. To our knowledge, this dataset is currently the most extensive collection of kernel vulnerabilities with workable PoCs. Significantly, this subset falls within the 1,829 vulnerabilities for which vulnerable versions were identified.

Dataset Availability. To facilitate ongoing research in the field of Linux kernel vulnerabilities, we will open-source the comprehensive dataset, encompassing vulnerabilities, their respective vulnerable versions, associated configs, and operational PoCs.

5.2 Performance in Reproduction

For each vulnerability in the 66-CVE dataset, we run KernJC to build the vulnerable environments and then compile and execute the associated PoC in the target environments for reproduction. Initially, we set $N = 1$ for the N-hop setting within the discovery of *HSC* and *HDC*. For each vulnerability with the source code downloaded, we run KernJC twice to build two environments

Table 3: Vulnerability Config Identification Statistics. The value in the Kernel column is the kernel source code version on which the configs are identified. Abbreviations are: configs in vulnerability descriptions (*DDC*), path-level configs (*DPC*), code-level configs (*DCC*), configs that are reachable from any direct config (*HRC*), configs holding one-hop select relation to any direct config (*HSC*), configs holding one-hop depend relation to any direct config (*HDC*). Underlined numbers indicate that one or more configs in the column (*HSC*/*HDC*) are needed to activate the related vulnerability.

CVE	Subsystem	Kernel	Kconfig Graph	DDC	DPC	DCC	HRC	HSC	HDC
CVE-2016-10150	virt/kvm	v4.8.12	12337v+38250e	0	1	0	39	0	4
CVE-2016-4557	kernel/bpf	v4.5.4	11845v+36556e	0	1	0	0	2	0
CVE-2016-6187	security/apparmor	v4.6.4	12021v+37152e	0	1	0	14	0	2
CVE-2017-16995	kernel/bpf	v4.14.8	13436v+41928e	0	1	0	0	2	0
CVE-2019-6974	virt/kvm	v4.20.7	14054v+44510e	0	1	0	42	0	4
CVE-2020-27194	kernel/bpf	v5.8.14	15340v+50234e	0	1	0	0	2	1
CVE-2020-8835	kernel/bpf	v5.6	14957v+48344e	0	1	0	0	2	1
CVE-2021-22555	net/netfilter	v5.11.14	15808v+51956e	0	7	1	10	3	406
CVE-2021-34866	kernel/bpf	v5.13.13	15982v+52565e	0	1	0	0	2	3
CVE-2021-3490	kernel/bpf	v5.12.3	15855v+52142e	0	1	0	0	2	2
CVE-2021-3573	net/bluetooth	v5.12.9	15851v+52148e	0	1	0	32	0	45
CVE-2022-1015	net/netfilter	v5.16.17	16244v+53665e	0	1	0	4	0	241
CVE-2022-25636	net/netfilter	v5.16.11	16243v+53663e	0	4	0	19	2	241
CVE-2022-32250	net/netfilter	v5.18.1	16542v+54754e	0	1	0	4	0	238
CVE-2022-34918	net/netfilter	v5.18.10	16548v+54770e	0	1	0	4	0	238
CVE-2023-32233	net/netfilter	v6.3.1	17120v+57166e	0	2	0	5	0	317

with different config identification approaches: (1) for the first one, KernJC builds the environment with the identified configs using the aforementioned graph-based approach, and (2) for the second one, KernJC builds it with the default configs (`make defconfig`) as the baseline. The PoC will be executed in both of these two environments. After the PoC is executed, we observe whether a KASAN report is generated in the kernel log or the special OS state described in the PoC file is achieved, to determine the result of reproduction.

We conduct extra reproduction experiments for each vulnerability with inaccurate version claims detected to ensure the correctness of the version identification process. Specifically, we run KernJC to build a reproduction environment with a detected inaccurate version and KernJC-identified configs (a combination of identified non-default configs and the default configs from `make defconfig`) for each case and execute the PoC within it.

Vulnerabilities that can only be reproduced with the activation of KernJC-identified configs or have false positive version claims detected by KernJC are listed in Table 2. The remaining results can be found in Table 6 of Appendix B. Results show that KernJC accomplishes the reproduction for all 66 vulnerabilities, indicating that it has effectively created a vulnerable environment for each of them.

Among these vulnerabilities, 32 of 66 (48.5%) need non-default configs identified by KernJC to be activated, and 34 of 66 (51.5%) can be activated by default configs. Additionally, 4 of 66 (6.1%) are detected to have false positive version claims in NVD. Interestingly, 2 of the 4 false positive cases (CVE-2021-22555 and CVE-2021-3573) are from the 32 vulnerabilities relying on non-default configs, and the other 2 vulnerabilities (CVE-2020-14381 and CVE-2020-25656) can be activated by default configs. As shown in Table 2, in summary, 34 of 66 (51.5%) vulnerabilities cannot be intuitively reproduced with claimed versions and default configs, due to either false positive version claims in NVD or extra non-default configs to activate the vulnerabilities. Among the 34 vulnerabilities, the "kernel/bpf" and "net/netfilter" subsystems contribute the largest number of vulnerabilities, which is 6 for both of them, compared with other subsystems in the table.

To better understand the usableness of KernJC, we also measure the time consumption of version and confi identification during the experiments. On average, given a vulnerability, it takes KernJC 3.91 seconds to identify a vulnerable version and 9.16 seconds to identify the vulnerability configs. This time cost is accepted, which significantly reduces the manual effort.

5.3 Role of Identified Configs

As shown in Figure 8, the vulnerability configs identified by KernJC are categorized into six categories: configs in vulnerability descriptions (*DDC*), path-level configs (*DPC*), code-level configs (*DCC*), configs recursively reachable from direct configs (*HRC*), configs with a one-hop `select` relationship to any direct config (*HSC*), and configs with a one-hop `depend` relationship to any direct config (*HDC*). KernJC can efficiently identify *DDC*, *DPC*, *DCC*, *HRC*, and *HSC* through patch parsing and graph analysis. Comparatively, as demonstrated in § 2.2, the discovery of *HSC* and *HDC* is more challenging but essential to activating vulnerabilities in complex subsystems. To this end, we analyze the 32 vulnerabilities relying on non-default configs, categorizing the identified configs and examining the relevance and contribution of *HSC* and *HDC* categories.

Vulnerabilities that rely on *HSC* or *HDC* are presented in Table 3, and results for the remaining vulnerabilities can be found

Table 4: Vulnerabilities with FP Version Range Claims in NVD. Vulnerable Version is the first vulnerable version downawards adjacent to the lower boundary of the FP version range. FP Count is the number of FP versions within the FP version range. Abbreviation: FP = False Positive.

CVE	CVSS	FP Version Range	Vulnerable Version	FP Count
CVE-2017-1000407	7.4	v4.14.6 – v4.14.325	v4.14.5	320
CVE-2017-18216	5.5	v4.14.57 – v4.14.325	v4.14.56	269
CVE-2017-18224	4.7	v4.14.57 – v4.14.325	v4.14.56	269
CVE-2020-35508	4.5	v5.9.7 – v5.11.22	v5.9.6	229
CVE-2021-4002	4.4	v5.15.5 – v5.15.132	v5.15.4	128
CVE-2021-4090	7.1	v5.15.5 – v5.15.132	v5.15.4	128
CVE-2022-0264	5.5	v5.15.11 – v5.15.132	v5.15.10	122
CVE-2021-4155	5.5	v5.15.14 – v5.15.132	v5.15.13	119
CVE-2016-10906	7.0	v4.4.191 – v4.4.302	v4.4.190	112
CVE-2015-4170	4.7	v3.12.7 – v3.13.3	v3.12.6	72

in Table 7 of Appendix B. The results indicate that half of these vulnerabilities (16 out of 32) necessitate *HSC* or *HDC* for activation. Consequently, *HSC* and *HDC* identified by KernJC play an important role in constructing effective reproduction environments for kernel vulnerabilities.

It should be noted that not all of the identified *HSC* and *HDC* contribute to the activation of a specific vulnerability. For some vulnerabilities, especially those from the Netfilter subsystem, the number of exploited *HDC* on average tends to be 280, due to the intricate dependencies of the subsystem, while only a small set of these configs actually matters. The effective *HDC* for these vulnerabilities are analyzed in § 5.5.1. This phenomenon confirms the necessity of the one-hop design for *HSC* and *HDC* identification. However, we argue that such config identification results do not affect the efficacy of KernJC, as demonstrated by the reproduction results in Table 2. Besides, according to Table 7, on average, the Kconfig graph built by KernJC has 14,824 vertexes and 47,950 edges, which highlights the complexity of kernel and implies time-consuming work to handle the configs manually. Compared to the scale of this graph, the 280 configs introduced in *HDC* are trivial.

Additionally, only in the case of CVE-2017-18344 were configs identified in the vulnerability description (*DDC*), suggesting that descriptions are less impactful than other sources, such as patches and source code, in the config identification process. This fact also implies that it is infeasible for analysts to directly figure out the necessary configs for reproduction by retrieving the vulnerability descriptions.

5.4 Detection of False Positive Version Claims

The accurate selection of vulnerable versions is crucial for the effective reproduction of kernel vulnerabilities. Wrong versions can mislead analysts and result in significant time wastage. Thus, we aim to assess the proficiency of KernJC in identifying false positive version range claims in the NVD database for upstream Linux kernel vulnerabilities.

The whole CVE dataset mentioned in § 5.1 is used for this evaluation. KernJC's approach involves mapping the claimed vulnerable version ranges into a list of kernel release versions, followed by a downward analysis starting from the upper boundary to identify false positives. This process continues until a genuine vulnerable version is found or the lower boundary is reached, with each false positive instance recorded. Lastly, KernJC compiles and reports the total count of false positive versions detected.

Table 5: HDC Needed by Vulnerabilities in Netfilter Subsystem. The `CONFIG_` prefix is omitted for clearness.

CVE	HDC
CVE-2021-22555	`NETFILTER_XT_TARGET_NFQUEUE`
CVE-2022-1015	`NF_TABLES, NF_TABLES_IPV4`
CVE-2022-25636	`NF_TABLES, NF_TABLES_NETDEV`
CVE-2022-32250	`NF_TABLES, NF_TABLES_IPV4`
CVE-2022-34918	`NF_TABLES, NF_TABLES_INET`
CVE-2023-32233	`NF_TABLES, NF_TABLES_INET, NFT_LOG, NFT_QUOTA`

The findings reveal that KernJC identifies false positive version range claims for 128 kernel vulnerabilities within the NVD database. The aggregate count of false positive versions is 3,042, averaging 24 false positive versions per identified vulnerability. The comprehensive table of these identification results (Table 8) is accessible in Appendix B, with the top 10 vulnerabilities sorted by false positive version count presented in Table 4.

5.5 Case Studies

5.5.1 Vulnerabilities in BPF & Netfilter Subsystems . Within the evaluation results presented in Table 6, we observe that two subsystems, eBPF and Netfilter, occur more often than other subsystems, each contributing six vulnerabilities to the dataset of 66 CVEs. Typically, more complex systems are susceptible to a higher number of vulnerabilities. In turn, the prevalence of vulnerabilities in eBPF and Netfilter underscores the complexity of these subsystems, a conclusion further corroborated in § 5.3. Analysis of config-dependent vulnerabilities (Table 7) reveals that most vulnerabilities associated with *HSC* or *HDC* configs originate from these two subsystems, indicating the increased challenges in manually constructing reproduction environments, while KernJC succeeds in identifying these configs and generating the environments for these vulnerabilities. Considering this, we investigate the *HSC* and *HDC* config of these twelve vulnerabilities in eBPF and Netfilter subsystems to precisely uncover their config dependencies.

For example, although the basic `CONFIG_BPF` (*DPC*) necessitated by the six eBPF vulnerabilities (CVE-2016-4557, CVE-2017-16995, CVE-2020-27194, CVE-2020-8835, CVE-2021-34866, and CVE-2021-3490), as indicated by file dependencies in the Makefile, is easy to figure out with manual analysis, the examination of the six reveals a consistent requirement for the `CONFIG_BPF_SYSCALL` (*HSC* identified by KernJC) to be enabled for their activation, which is comparatively non-intuitive. The absence of `CONFIG_BPF_SYSCALL` will lead to wasted time and failed reproduction.

The analysis of the six Netfilter vulnerabilities reveals varied config requirements for each, along with their associated PoCs, complicating the manual construction of reproduction environments. Utilizing KernJC's identification results, we conducted a manual analysis to determine the minimal config set of *HSC* and *HDC* for each vulnerability. Our findings indicate that the effective hidden configs

for these vulnerabilities are exclusively derived from their respective *HDC* sets, as detailed in Table 5. Specifically, we observed that the official announcement for CVE-2022-32250 [3] only cited the `CONFIG_NETFILTER` and `CONFIG_NF_TABLES` configs as necessary for activating this vulnerability. However, our reproduction demonstrated the additional requirement of `CONFIG_NF_TABLES_IPV4`. Similarly, the disclosure for CVE-2023-32233 [42] did not include the essential `CONFIG_NFT_LOG` and `CONFIG_NFT_QUOTA` config. The absence of these configs, as identified by KernJC, results in the failure of the original PoCs provided in these announcements during our reproduction experiments.

5.5.2 Cross-minor-version FP: CVE-2018-1000028 . Within the comprehensive list of FP results detailed in Appendix B, CVE-2018-1000028 [61] is particularly noteworthy. This vulnerability, stemming from incorrect access control in the kernel's NFS server implementation, potentially allows remote attackers to read or write files via NFS inappropriately.

Our analysis highlights that CVE-2018-1000028 is the sole vulnerability within this list characterized by both a high CVSS severity score (7.4) and FP version ranges spanning two minor Linux release versions: NVD claims the vulnerable version range spans from v4.14.8 (inclusive) to v4.14.23 (inclusive) and from v4.15.1 (inclusive) to v4.15.7 (inclusive). However, our detection indicates that patches were applied from v4.14.16 (inclusive) to v4.14.23 (inclusive) and from v4.15.1 (inclusive) to v4.15.7 (inclusive), thereby reducing the scope of vulnerability. Drawing upon this analysis, we successfully reproduced the vulnerability in the v4.14.15 kernel version. Conversely, our attempts to reproduce it in the v4.14.16 kernel version were unsuccessful.

For vulnerabilities like CVE-2018-1000028 that have incorrect version claims spanning more than one minor version number (*e.g.,* v4.14 and v4.15 for CVE-2018-1000028) of kernel release versions in online databases, the impact is even more serious, as it is more difficult for analysts to select a vulnerable version for effective reproduction accurately. In such cases, KernJC can help save much time and effort on the pre-reproduction work.

6 DISCUSSION

6.1 Reproducibility and Exploitability

Reproducibility and exploitability serve as vital metrics for gauging the severity of vulnerabilities and the associated risk in potentially susceptible environments. KernJC's methodology aids in the reproducibility assessment of Linux kernel vulnerabilities by creating an environment where the specific vulnerability is both present and accessible from user space. However, the environments generated by KernJC may not fulfill the prerequisites for successful exploitation. For instance, to reliably **exploit** vulnerabilities in a modern Linux kernel, where various mitigation techniques are in place, many exploitations utilize the Filesystem in Userspace (FUSE) [55] or userfaultfd [17]. These mechanisms are contingent on the activation of non-default configs: `CONFIG_FUSE_FS` for FUSE and `CONFIG_USERFAULTFD` for userfaultfd. Since these configs are not related to the activation of vulnerabilities, their identification falls beyond the scope of this paper, which we intend to explore in future work.

6.2 Dataset Preparation

The 66-CVE dataset used in § 5.2, detailed in Table 2 and more extensively in Table 6 of Appendix B, spans over nine years and encapsulates a wide array of vulnerability types within the Linux kernel's diverse subsystems. This comprehensive coverage underpins our assertion that the dataset is representative, and we anticipate that KernJC will demonstrate effective performance on other kernel vulnerabilities not explicitly covered in this paper. Time and labor constraints preclude a broader evaluation of KernJC against all existing vulnerabilities in upstream Linux kernels. Considering the complexity and rapid evolution of the Linux kernel ecosystem, there could probably exist corner cases where KernJC fails to build effective reproduction environments for some vulnerabilities not mentioned in this paper. To facilitate future kernel vulnerability reproduction work, we plan to open-source KernJC and engage with the community for ongoing iterative evaluation and enhancement.

6.3 Vulnerability Profiling

Currently, KernJC leverages the NVD, which is in sync with the official MITRE CVE database [56], to gather essential information necessary for reconstructing environments for Linux kernel vulnerabilities. We acknowledge that if information is missing from these databases, the data collected may not suffice to accurately recreate such vulnerable environments. However, the occurrence of absent information is exceedingly rare [18] and does not significantly impact the generalizability or effectiveness of KernJC. Furthermore, as discussed in § 3.2, KernJC is highly extensible. It can swiftly respond to updates in CVE information, allowing it to quickly adapt and create the appropriate environment.

Additionally, the availability and quality of patches pose challenges to KernJC's ability to accurately identify the correct kernel version for reproducing vulnerable environments. In terms of patch availability, our analysis of all CVEs affecting the upstream kernel [54] reveals that patches are available for 95.7% of vulnerabilities in upstream versions. Regarding the quality of these patches, which is beyond the scope of our study of KernJC, one may consult existing literature [79], to understand how to extract information from existing patches. To gather as much information on CVE and patches as possible for better profiling of kernel vulnerabilities, a viable approach involves sourcing data from the Linux kernel mailing list [53] or the kernel Bugzilla [48]. This approach represents an engineering enhancement, and we aim to refine our vulnerability profiling methodology in future development and iterations of the open-source project of KernJC.

6.4 Other Influence Factors of Reproduction

Although KernJC is able to generate effective environments where kernel vulnerabilities can be triggered, there could be other factors influencing the success of reproduction. For example, the PoC programs may need to do specific preliminary operations to prepare the system state for vulnerability triggering, *e.g.,* namespace switching, device initialization, and filesystem mounting, as demonstrated by the syzkaller project [22]. Such setup operations heavily rely on the characteristics of specific vulnerabilities. We leave the discovery and enforcement of these setups as future work.

7 RELATED WORK

Reproducibility Assessment. Mu *et al.* [57] conducted the first empirical analysis encompassing both the vulnerable environment and the PoC program. Chen *et al.* [13] introduced a binary similarity-based method to deduce the specific build configurations for vulnerabilities in userland programs. Additionally, various studies [9, 12, 83] have investigated the automated generation of PoC using vulnerability-related data, such as descriptions, source code, and patches. Pham *et al.* [67] utilized symbolic execution for generating PoC for binaries, with a notable focus by You *et al.* [83] on the automated generation of PoC for Linux kernel vulnerabilities. Avgerinos *et al.* [4] pioneered the concept of Automatic Exploit Generation (AEG). Subsequent works [2, 6, 27, 30, 66, 73, 74, 87] have expanded AEG to various vulnerability types, including data-oriented, web-based, and heap-based vulnerabilities. Specifically, in the context of Linux kernel vulnerabilities, significant contributions [14, 33, 50, 77, 78] have been made toward AEG for vulnerabilities of certain types.

Vulnerability Version Assessment. Constructing the vulnerable environment necessitates identifying a kernel version with known vulnerabilities. However, the accuracy of version information from online vulnerability databases is not always guaranteed. A viable method to ascertain the vulnerability of a specific kernel version involves examining whether the code manifesting the vulnerability or its associated patch exists within the target source code or the final compiled programs. Nguyen *et al.* [59] employed a technique to detect vulnerable versions by verifying the presence of a known vulnerability's code in its preceding versions. Bao *et al.* [5] introduced the V-SZZ algorithm, designed to validate vulnerable versions across 172 CVEs from 55 C/C++ or Java projects, by pinpointing the earliest commit that altered the lines of code in question. Further, Jiang *et al.* [34] and Zhang *et al.* [88] concentrated on testing the presence of patches in downstream kernel image binaries. Dong *et al.* [18] proposed VIEM, a model utilizing deep learning to identify discrepancies, such as those in vulnerable version listings, between the NVD database, unstructured CVE descriptions, and external reports. Nonetheless, these studies above do not specifically target the upstream versions of Linux kernel source code. While VIEM effectively detects inconsistencies, it does not discern which piece of conflicting information is accurate.

Linux Kconfig Analysis. Kernel configs represent another crucial element of the vulnerable environment. The complexity inherent in the Kconfig mechanism of modern kernels, characterized by an extensive array of configs, renders the identification of specific configs responsible for introducing vulnerabilities a non-trivial task. To date, no research directly addresses this specific challenge. However, several studies within the software engineering domain offer valuable insights into understanding and managing Kconfig complexities. Notable works include the analysis and remediation of Kconfig defects by Franz *et al.* [19], the evaluation of Kconfig models by Hengelein *et al.* [28], the detection of configuration-related errors by Oh *et al.* [65], and the lightweight extraction of variability information by Ruprecht *et al.* [69].

8 CONCLUSION

This paper introduced KernJC, an innovative tool designed to automate the generation of vulnerable environments for Linux kernel vulnerabilities. KernJC stands out for its patch-based vulnerable version detection capability, enabling it to discern and rectify erroneous version information in NVD. This functionality is instrumental in accurately identifying the genuinely vulnerable version of a given Linux kernel vulnerability. Additionally, KernJC leverages both direct and hidden config information within the Kconfig graph, ensuring the activation of the target vulnerability within the constructed environment. We evaluate KernJC with real-world kernel vulnerabilities collected from existing research. The evaluation results underscore the KernJC 's proficiency in accurately pinpointing the genuine vulnerable versions and requisite kernel configs. More importantly, KernJC demonstrates its capability to reliably establish vulnerable environments conducive to the reproduction of kernel vulnerabilities.

ACKNOWLEDGMENTS

We would like to thank Kaihang Ji, Jianing Wang, Anis Bin Yusof, and the anonymous reviewers for their valuable comments. This research is supported by the National Research Foundation, Singapore, through the National Cybersecurity R&D Lab at the National University of Singapore under its National Cybersecurity R&D Programme (Award No. NCR25-NCL P3-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and National Cybersecurity R&D Lab at the National University of Singapore.

REFERENCES

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. {SHARD}:{Fine-Grained} Kernel Specialization with {Context-Aware} Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*. 2435–2452.

[2] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 641–652.

[3] Hugues ANGUELKOV. [n. d.]. Re: Linux kernel: Netfilter heap buffer overflow in nft_set_elem_init. https://seclists.org/oss-sec/2022/q3/2.

[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:14420062

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

[6] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 824–839.

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

[8] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. 2022. Midas: Systematic Kernel {TOCTTOU} Protection. In *31st USENIX Security Symposium (USENIX Security 22)*. 107–124.

[9] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 143–157.

[10] bsauce. 2022. Comments for CVE-2021-22555. https://github.com/bsauce/blog-comment/issues/23.

[11] bsauce. 2022. Comments for CVE-2022-0185. https://github.com/bsauce/blog-comment/issues/28.

[12] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. 322–335.

[13] Ligeng Chen, Jian Guo, Zhongling He, Dongliang Mu, and Bing Mao. 2021. Robin: Facilitating the reproduction of configuration-related vulnerability. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 91–98.

[14] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. {KOOBE}: towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*. 1093–1110.

[15] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1165–1184.

[16] Yueqi Chen and Xinyu Xing. 2019. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1707–1722.

[17] Vincent Dehors. [n. d.]. Exploitation of a double free vulnerability in Ubuntu shiftfs driver (CVE-2021-3492). http://bit.ly/46U6Zjm.

[18] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*. 869–885.

[19] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. Configfix: interactive configuration conflict resolution for the linux kernel. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 91–100.

[20] Google. 2022. Linux: Heap Out-Of-Bounds Write in xt_compat_target_from_user. https://github.com/google/security-research/security/advisories/GHSA-xxx5-8mvq-3528.

[21] Google. 2024. GitHub - google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[22] Google. 2024. syzkaller/executor/common_linux.h. https://github.com/google/syzkaller/blob/master/executor/common_linux.h.

[23] Google. 2024. syzkaller/tools/create-image.sh. https://github.com/google/syzkaller/blob/master/tools/create-image.sh.

[24] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3262–3278.

[25] Patrick McHardy Harald Welte. [n. d.]. net/netfilter/x_tables.c. https://bit.ly/4aslkH0.

[26] Red Hat. 2024. Red Hat Bugzilla Main Page. https://bugzilla.redhat.com/.

[27] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1689–1706.

[28] Stefan Hengelein and Daniel Lohmann. 2015. *Analyzing the Internal Consistency of the Linux KConfig Model*. Ph. D. Dissertation. Master's thesis. University of Erlangen, Dept. of Computer Science.

[29] Jamie Hill-Daniel. 2022. kernel/git/torvalds/linux.git. https://bit.ly/4aknIPN.

[30] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of {Data-Oriented} Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*. 177–192.

[31] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[32] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SEGFUZZ: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2104–2121.

[33] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2023. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2122–2137.

[34] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1149–1163.

[35] Jesper Juhl. 2024. Applying Patches To The Linux Kernel; The Linux Kernel documentation. https://www.kernel.org/doc/html/next/process/applying-patches.html.

[36] kashyapc.fedorapeople.org. [n. d.]. Snapshots Handout. https://kashyapc.fedorapeople.org/virt/lc-2012/snapshots-handout.html.

[37] kernel.org. 2014. Kbuild; The Linux Kernel documentation. https://docs.kernel.org/kbuild/kbuild.html.

[38] kernel.org. 2024. Kconfig Language; The Linux Kernel documentation. https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html.

[39] kernel.org. 2024. Kernel.org. https://git.kernel.org/.

[40] kernel.org. 2024. Submitting patches: the essential guide to getting your code into the kernel; The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/process/submitting-patches.html

[41] Michael Kerrisk. [n. d.]. proc(5). https://man7.org/linux/man-pages/man5/proc.5.html.

[42] Piotr Krysiuk. [n. d.]. Re: [CVE-2023-32233] Linux kernel use-after-free in Netfilter nf_tables when processing batch requests can be abused to perform arbitrary reads and writes in kernel memory. https://www.openwall.com/lists/oss-security/2023/05/15/5.

[43] Michael Larabel. 2021. Linux 5.12 Coming In At Around 28.8 Million Lines, AMDGPU Driver Closing In On 3 Million. https://www.phoronix.com/news/Linux-5.12-rc1-Code-Size.

[44] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. 2023. Pspray: Timing {Side-Channel} based Linux Kernel Heap Exploitation Technique. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6825–6842.

[45] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. 2363–2380.

[46] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4211–4228.

[47] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1963–1976.

[48] Linux. 2024. Bugzilla. https://bugzilla.kernel.org/.

[49] Linux. 2024. Index of /pub/linux/kernel/. https://mirrors.edge.kernel.org/pub/linux/kernel/.

[50] Danjun Liu, Pengfei Wang, Xu Zhou, and Baosheng Wang. 2022. ERACE: Toward Facilitating Exploit Generation for Kernel Race Vulnerabilities. *Applied Sciences* 12, 23 (2022), 11925.

[51] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. 2022. From Release to Rebirth: Exploiting Thanos Objects in Linux Kernel. *IEEE Transactions on Information Forensics and Security* 18 (2022), 533–548.

[52] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. {LinKRID}: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *31st USENIX Security Symposium (USENIX Security 22)*. 125–142.

[53] lkml.org. 2024. LKML.ORG. https://lkml.org/.

[54] Nicholas Luedtke. 2024. Linux Kernel CVEs. https://www.linuxkernelcves.com/.

[55] LukeGix. [n. d.]. FUSE for Linux Exploitation 101. https://exploiter.dev/blog/2022/FUSE-exploit.html.

[56] MITRE. 2024. CVE. https://cve.mitre.org/.

[57] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.

[58] NetworkX. 2014. NetworkX. https://networkx.org/.

[59] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21 (2016), 2268–2297.

[60] NVD. 2017. NVD - CVE-2017-18344. https://nvd.nist.gov/vuln/detail/CVE-2017-18344.

[61] NVD. 2018. NVD - CVE-2018-1000028. https://nvd.nist.gov/vuln/detail/CVE-2018-1000028.

[62] NVD. 2022. NVD - CVE-2022-0185. https://nvd.nist.gov/vuln/detail/CVE-2022-0185

[63] NVD. 2022. NVD CVE-2021-22555. https://nvd.nist.gov/vuln/detail/CVE-2021-22555.

[64] NVD. 2024. National Vulnerability Database. https://nvd.nist.gov/.

[65] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding broken Linux configuration specifications by statically analyzing the Kconfig language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 893–905.

[66] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. 2022. {FUGIO}: Automatic Exploit Generation for {PHP} Object Injection Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 197–214.

[67] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 891–901.

[68] Pansilu Pitigalaarachchi, Xuhua Ding, Haiqing Qiu, Haoxin Tu, Jiaqi Hong, and Lingxiao Jiang. 2023. KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2009–2023.

[69] Andreas Ruprecht. 2015. *Lightweight Extraction of Variability Information from Linux Makefiles*. Ph. D. Dissertation. Citeseer.

[70] Ubuntu. 2024. Security. https://ubuntu.com/security.

[71] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. 2018. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 374–391.

[72] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. 2023. {AlphaEXP}: An Expert System for Identifying {Security-Sensitive} Kernel Objects. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4229–4246.

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

[74] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. 2021. {MAZE}: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*. 1647–1664.

[75] Zicheng Wang, Yueqi Chen, and Qingkai Zeng. 2023. {PET}: Prevent Discovered Errors from Being Triggered in the Linux Kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4193–4210.

[76] Florian Westphal. 2022. kernel/git/torvalds/linux.git. https://bit.ly/3NnVtpF.

[77] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. 1187–1204.

[78] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. {FUZE}: Towards Facilitating Exploit Generation for Kernel {Use-After-Free} Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 781–797.

[79] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating Security Risks in Linux with {KLAUS}: A Method for Evaluating Patch Correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4247–4264.

[80] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. 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*. 414–425.

[81] Yutian Yang, Wenbo Shen, Bonan Ruan, Wenmao Liu, and Kui Ren. 2021. Security challenges in the container cloud. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 137–145.

[82] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. {In-Kernel}{Control-Flow} Integrity on Commodity {OSes} using {ARM} Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. 89–106.

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

[84] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2849–2866.

[85] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 71–88. https://www.usenix.org/conference/usenixsecurity22/presentation/zeng

[86] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2023. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3093–3107. https://doi.org/10.1145/3576915.3623220

[87] Bin Zhang, Jiongyi Chen, Runhao Li, Chao Feng, Ruilin Li, and Chaojing Tang. 2023. Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation {Distance-Guided} Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4499–4515.

[88] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. 887–902.

[89] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. {StateFuzz}: System {Call-Based}{State-Aware} Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3273–3289.

A AUTOMATED VULNERABILITY CONFIG IDENTIFYING

The entire algorithm of the graph-based approach for identifying necessary kernel configs is shown in Algorithm 2. GetVulConfigs is the entrance of this algorithm.

Algorithm 2: Vulnerability Config Identification

Input: V: Vulnerability Description; P: Patch Text for V; SC: Vulnerable Kernel Source Code
Output: S: Set of Identified Configs

Procedure *GetVulConfigs*(V, P, SC)
  D = GetDirectConfigs(V, P, SC)
  G = BuildKconfigGraph(SC)
  H = GetHiddenConfigs(D, G)
  $S = D \cup H$
  return S

Procedure *GetDirectConfigs*(V, P, SC)
  DF = affected files mentioned in V (optional)
  DDC = configs mentioned in V (optional)
  DPC = configs in SC to enable DF and files in P
  DCC = configs in SC to enable code in P by #ifdef
  return $DDC \cup DPC \cup DCC$

Procedure *BuildKconfigGraph*(SC)
  G = empty graph
  RK = root Kconfig file in SC
  AddKconfigNodes(G, RK, SC)
  AddKconfigEdges(G, RK, SC)
  return G

Procedure *GetHiddenConfigs*(D, G)
  $H = \emptyset$
  foreach *c in D* **do**
    HRC = reachable configs from c in G
    HSC = configs with *select* relation to c in G
    HDC = configs with *depend* relation to c in G
    Add HRC, HSC, HDC into H
  end
  return H

Procedure *AddKconfigNodes*(G, K, SC)
  foreach *line in K* **do**
    if *line defines config c* **then**
      Add c into G
    end
    if *line imports new Kconfig file nk* **then**
      AddKconfigNodes(G, nk, SC)
    end
  end

Procedure *AddKconfigEdges*(G, K, SC)
  foreach *line in K* **do**
    if *line defines config c* **then**
      Add edges from/to c into G
    end
    if *line defines any block from* $(menu, if, choice)$ **then**
      Add edges derived from the block into G
    end
    if *line imports new Kconfig file nk* **then**
      AddKconfigEdges(G, nk, SC)
    end
  end

B ADDITIONAL EVALUATION RESULTS

Performance in Vulnerability Reproduction. The evaluation results for reproducing the 66 vulnerabilities are presented in Table 6. Results indicate that KernJC effectively creates a vulnerable environment for all 66 vulnerabilities. Of the 66 vulnerabilities, 32 vulnerabilities (48.5%) require non-default configs identified by KernJC to be activated, whereas 34 vulnerabilities (51.5%) can be activated by default configs. Furthermore, 4 vulnerabilities (6.1%) have been identified to have false positive version claims in NVD.

Role of Configs Identified by KernJC. The evaluation results for analyzing the role of identified vulnerability configs are presented in Table 7. Out of the 32 vulnerabilities, 16 require *HSC* or *HDC* configs to be activated. The Kconfig graph constructed by KernJC has an average of 14,824 vertices and 47,950 edges, which indicates the intricate nature of the Linux kernel. This complexity also implies that manual handling of the configs can be a time-consuming process.

Detection of False Positive Version Claims. The evaluation results for false positive version detection on the 2,256 vulnerabilities are presented in Table 8. The findings reveal that KernJC identifies false positive version range claims in 128 out of 2,256 (5.7%) kernel vulnerabilities within the NVD database. The aggregate count of false positive versions is 3,042, which averages to 24 false positive versions per identified vulnerability.

Table 6: Vulnerability Reproduction Results. "RwKC?" and "RwDC?" denote reproducibility with KernJC-identified and default configs, respectively. "FPV?" indicates false positive version claims in NVD. Blue CVE IDs signify the need for non-default configs or presence of false positive claims. Abbreviations: UAF = Use after Free; OOB = Out of Bounds; TOCTOU = Time of Check to Time of Use; DF = Double Free; ND = Null-pointer Dereference.

CVE	Type	CVSS	Subsystem	RwKC?	RwDC?	FPV?	Paper(s)
CVE-2015-3636	UAF	4.9	net/ipv4	✓	✓	✗	SLAKE
CVE-2016-0728	UAF	7.8	security/keys	✓	✓	✗	K(H)eaps, LinKRID, SHARD, ELOISE, SLAKE, RetSpill
CVE-2016-10150	UAF	9.8	virt/kvm	✓	✗	✗	K(H)eaps, ELOISE, SLAKE, Kepler
CVE-2016-4557	UAF	7.8	kernel/bpf	✓	✗	✗	AEM, K(H)eaps, ELOISE, SLAKE, Kepler, RetSpill
CVE-2016-6187	OOB	7.8	security/apparmor	✓	✗	✗	Pspray, PET, AEM, K(H)eaps, ELOISE, KOOBE, SLAKE, Kepler, RetSpill
CVE-2016-6516	TOCTOU	7.4	fs/ioctl	✓	✓	✗	Midas
CVE-2016-8655	UAF	7.8	net/packet	✓	✓	✗	SegFuzz, AEM, K(H)eaps, ExpRace, SLAKE, Kepler, Razzer
CVE-2016-9793	OOB	7.8	net/core	✓	✓	✗	AEM, Kepler
CVE-2017-1000112	OOB	7.0	net/ipv4, net/ipv6	✓	✓	✗	AEM, ELOISE, KOOBE, SLAKE
CVE-2017-10661	UAF	7.0	fs/timefd	✓	✓	✗	AEM, K(H)eaps, ELOISE, SLAKE, Kepler, RetSpill
CVE-2017-11176	UAF	7.8	ipc/mqueue	✓	✓	✗	AEM, K(H)eaps, RetSpill
CVE-2017-15265	UAF	7.0	sound/core	✓	✓	✗	ExpRace
CVE-2017-15649	UAF	7.8	net/packet	✓	✓	✗	SegFuzz, AEM, K(H)eaps, ELOISE, SLAKE, Kepler
CVE-2017-16995	Logic	7.8	kernel/bpf	✓	✗	✗	AEM, Kepler
CVE-2017-17052	UAF	7.8	kernel/fork	✓	✓	✗	ELOISE, SLAKE
CVE-2017-17053	UAF	7.0	asm/mmu_context	✓	✓	✗	ELOISE, SLAKE, Kepler
CVE-2017-17712	UAF	7.0	net/ipv4	✓	✓	✗	SegFuzz, ExpRace, Razzer
CVE-2017-18344	OOB	5.5	kernel/time	✓	✗	✗	PET, AEM
CVE-2017-2636	DF	7.0	drivers/tty	✓	✗	✗	SegFuzz, PET, AEM, K(H)eaps, ExpRace, ELOISE, SLAKE, Kepler, Razzer, RetSpill
CVE-2017-5123	Logic	8.8	kernel/exit	✓	✓	✗	Hybrid, AEM, SHARD, Kepler
CVE-2017-6074	DF	7.8	net/dccp	✓	✗	✗	Pspray, AEM, K(H)eaps, ELOISE, SLAKE, Kepler, RetSpill
CVE-2017-7184	OOB	7.8	net/xfrm	✓	✓	✗	Pspray, PET, AEM, K(H)eaps, ELOISE, KOOBE, SLAKE, Kepler, RetSpill
CVE-2017-7308	OOB	7.8	net/packet	✓	✓	✗	PET, AEM, K(H)eaps, PAL, SHARD, ELOISE, KOOBE, SLAKE, Kepler, RetSpill
CVE-2017-7533	OOB	7.0	fs/notify	✓	✓	✗	SegFuzz, Pspray, K(H)eaps, ExpRace, ELOISE, KOOBE, RetSpill
CVE-2017-8824	UAF	7.8	net/dccp	✓	✗	✗	PET, AEM, K(H)eaps, SLAKE, Kepler, RetSpill
CVE-2017-8890	DF	7.8	net/ipv4	✓	✓	✗	AEM, K(H)eaps, ELOISE, SLAKE, Kepler
CVE-2018-10840	OOB	6.6	fs/ext4	✓	✓	✗	SLAKE
CVE-2018-12232	ND	5.9	net/socket	✓	✓	✗	SegFuzz
CVE-2018-12233	OOB	7.8	fs/jfs	✓	✗	✗	ELOISE
CVE-2018-18559	UAF	8.1	net/packet	✓	✓	✗	ELOISE, SLAKE
CVE-2018-5333	ND	5.5	net/rds	✓	✗	✗	AEM
CVE-2018-6555	UAF	7.8	net/irda	✓	✗	✗	Pspray, AlphaEXP, AEM, K(H)eaps, ELOISE, SLAKE, RetSpill
CVE-2019-15666	UAF	4.4	net/xfrm	✓	✓	✗	AlphaEXP, AEM, DirtyCred
CVE-2019-6974	UAF	8.1	virt/kvm	✓	✗	✗	SegFuzz, ExpRace
CVE-2020-14381	UAF	7.8	futex	✓	✓	✓	AlphaEXP
CVE-2020-14386	OOB	7.8	net/packet	✓	✓	✗	PET, DirtyCred
CVE-2020-16119	UAF	7.8	net/dccp	✓	✗	✗	PET, DirtyCred
CVE-2020-25656	UAF	4.1	drivers/tty	✓	✓	✓	DDRace
CVE-2020-25669	UAF	7.8	drivers/input	✓	✗	✗	StateFuzz
CVE-2020-27194	OOB	5.5	kernel/bpf	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2020-27830	ND	5.5	drivers/accessibility	✓	✗	✗	StateFuzz
CVE-2020-28097	OOB	5.9	vgacon	✓	✓	✗	StateFuzz
CVE-2020-28941	ND	5.5	drivers/accessibility	✓	✗	✗	StateFuzz
CVE-2020-8835	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred
CVE-2021-20226	UAF	7.8	io_uring	✓	✓	✗	AlphaEXP
CVE-2021-22555	OOB	7.8	net/netfilter	✓	✗	✓	PET, AlphaEXP, DirtyCred
CVE-2021-22600	DF	7.0	net/packet	✓	✓	✗	DirtyCred
CVE-2021-26708	UAF	7.0	net/vmw_vsock	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2021-27365	OOB	7.8	drivers/scsi	✓	✗	✗	Hybrid, DirtyCred, RetSpill
CVE-2021-33909	OOB	7.8	fs/seq_file.c	✓	✓	✗	AlphaEXP, DirtyCred
CVE-2021-34866	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred
CVE-2021-3490	OOB	7.8	kernel/bpf	✓	✗	✗	DirtyCred, RetSpill
CVE-2021-3573	UAF	6.4	net/bluetooth	✓	✗	✓	AlphaEXP
CVE-2021-41073	UAF	7.8	io_uring	✓	✓	✗	AlphaEXP, DirtyCred
CVE-2021-4154	UAF	8.8	kernel/cgroup	✓	✓	✗	PET, DirtyCred, RetSpill
CVE-2021-42008	OOB	7.8	drivers/net	✓	✗	✗	Hybrid, AlphaEXP, DirtyCred
CVE-2021-43267	OOB	9.8	net/tipc	✓	✗	✗	Hybrid, AlphaEXP, KRover, DirtyCred, PET, RetSpill
CVE-2022-0185	OOB	8.4	fs/fs_context	✓	✓	✗	PET, Hybrid, AlphaEXP, DirtyCred, RetSpill
CVE-2022-0995	OOB	7.8	watch_queue	✓	✗	✗	AlphaEXP, DirtyCred
CVE-2022-1015	OOB	6.6	net/netfilter	✓	✗	✗	PET
CVE-2022-1786	UAF	7.8	io_uring	✓	✓	✗	Hybrid, RetSpill
CVE-2022-24122	UAF	7.8	kernel/ucount	✓	✓	✗	DirtyCred
CVE-2022-25636	OOB	7.8	net/netfilter	✓	✗	✗	AlphaEXP, DirtyCred, RetSpill
CVE-2022-32250	UAF	7.8	net/netfilter	✓	✗	✗	Hybrid
CVE-2022-34918	OOB	7.8	net/netfilter	✓	✗	✗	PET, Hybrid
CVE-2023-32233	UAF	7.8	net/netfilter	✓	✗	✗	Hybrid

Table 7: Vulnerability Config Identification Statistics. The value in the Kernel column is the kernel source code version on which the configs are identified. Abbreviations are: configs in vulnerability descriptions (*DDC*), path-level configs (*DPC*), code-level configs (*DCC*), configs that are reachable from any direct config (*HRC*), configs holding one-hop select relation to any direct config (*HSC*), configs holding one-hop depend relation to any direct config (*HDC*). Underlined numbers indicate that one or more configs in the column (*HSC*/*HDC*) are needed to activate the related vulnerability.

CVE	Subsystem	Kernel	Kconfig Graph	DDC	DPC	DCC	HRC	HSC	HDC
CVE-2016-10150	virt/kvm	v4.8.12	12337v+38250e	0	1	0	39	0	4
CVE-2016-4557	kernel/bpf	v4.5.4	11845v+36556e	0	1	0	0	2	0
CVE-2016-6187	security/apparmor	v4.6.4	12021v+37152e	0	1	0	14	0	2
CVE-2017-16995	kernel/bpf	v4.14.8	13436v+41928e	0	1	0	0	2	0
CVE-2017-18344	kernel/time	v4.14.7	13436v+41929e	2	0	0	3	0	3
CVE-2017-2636	drivers/tty	v4.10.2	12706v+39472e	0	1	0	17	0	0
CVE-2017-6074	net/dccp	v4.9.12	12519v+38798e	0	1	0	9	0	0
CVE-2017-8824	net/dccp	v4.14.19	13441v+41941e	0	1	0	9	0	0
CVE-2018-12233	fs/jfs	v4.17.1	13588v+43147e	0	1	0	4	0	4
CVE-2018-5333	net/rds	v4.14.13	13437v+41930e	0	1	0	9	0	3
CVE-2018-6555	net/irda	v4.16.18	13626v+42836e	0	2	1	7	0	37
CVE-2019-6974	virt/kvm	v4.20.7	14054v+44510e	0	1	0	42	0	4
CVE-2020-16119	net/dccp	v5.8.10	15340v+50231e	0	1	0	5	0	0
CVE-2020-25669	drivers/input	v5.9.9	15456v+50684e	0	3	0	3	37	3
CVE-2020-27194	kernel/bpf	v5.8.14	15340v+50234e	0	1	0	0	2	1
CVE-2020-27830	drivers/accessibility	v5.9.13	15457v+50695e	0	2	0	19	0	0
CVE-2020-28941	drivers/accessibility	v5.9.9	15456v+50684e	0	2	0	19	0	0
CVE-2020-8835	kernel/bpf	v5.6	14957v+48344e	0	1	0	0	2	1
CVE-2021-22555	net/netfilter	v5.11.14	15808v+51956e	0	7	1	10	3	406
CVE-2021-26708	net/vmw_vsock	v5.10.12	15674v+51410e	0	1	0	4	0	6
CVE-2021-27365	drivers/scsi	v5.11.3	15808v+51950e	0	2	0	22	8	0
CVE-2021-34866	kernel/bpf	v5.13.13	15982v+52565e	0	1	0	0	2	3
CVE-2021-3490	kernel/bpf	v5.12.3	15855v+52142e	0	1	0	0	2	2
CVE-2021-3573	net/bluetooth	v5.12.9	15851v+52148e	0	1	0	32	0	45
CVE-2021-42008	drivers/net	v5.13.12	15981v+52560e	0	2	0	18	0	14
CVE-2021-43267	net/tipc	v5.14.15	16009v+52674e	0	1	0	5	0	4
CVE-2022-0995	watch_queue	v5.16.4	16244v+53661e	0	1	1	0	0	1
CVE-2022-1015	net/netfilter	v5.16.17	16244v+53665e	0	1	0	4	0	241
CVE-2022-25636	net/netfilter	v5.16.11	16243v+53663e	0	4	0	19	2	241
CVE-2022-32250	net/netfilter	v5.18.1	16542v+54754e	0	1	0	4	0	238
CVE-2022-34918	net/netfilter	v5.18.10	16548v+54770e	0	1	0	4	0	238
CVE-2023-32233	net/netfilter	v6.3.1	17120v+57166e	0	2	0	5	0	317

Table 8: Vulnerabilities with FP Version Range Claims in NVD. Vulnerable Version is the first vulnerable version downawards adjacent to the lower boundary of the FP version range. FP Count is the number of FP versions within the FP version range. Abbreviation: FP = False Positive.

CVE	CVSS	FP Version Range	Vulnerable Version	FP Count	CVE	CVSS	FP Version Range	Vulnerable Version	FP Count
CVE-2012-4444	5.0	v2.6.36 – v2.6.36	v2.6.35.14	1	CVE-2021-3744	5.5	v5.14.10 – v5.14.21	v5.14.9	12
CVE-2012-5375	4.0	v3.8 – v3.8	v3.7.10	1	CVE-2021-3753	4.7	v5.14.1 – v5.14.21	v5.14	21
CVE-2012-6536	2.1	v3.5.7 – v3.5.7	v3.5.6	1	CVE-2021-3764	5.5	v5.14.10 – v5.14.20	v5.14.9	11
CVE-2012-6538	1.9	v3.5.7 – v3.5.7	v3.5.6	1	CVE-2021-4002	4.4	v5.15.5 – v5.15.132	v5.15.4	128
CVE-2012-6542	1.9	v3.5.5 – v3.5.7	v3.5.4	3	CVE-2021-4090	7.1	v5.15.5 – v5.15.132	v5.15.4	128
CVE-2012-6545	1.9	v3.5.5 – v3.5.7	v3.5.4	3	CVE-2021-4155	5.5	v5.15.14 – v5.15.132	v5.15.13	119
CVE-2012-6647	4.9	v3.4.8 – v3.4.9	v3.4.7	2	CVE-2021-4203	6.8	v5.14.10 – v5.14.21	v5.14.9	12
CVE-2013-0217	5.2	v3.7.8 – v3.7.8	v3.7.7	1	CVE-2022-0264	5.5	v5.15.11 – v5.15.132	v5.15.10	122
CVE-2013-3224	4.9	v3.8.11 – v3.9	v3.8.10	4	CVE-2022-0322	5.5	v5.14.14 – v5.14.21	v5.14.13	8
CVE-2013-3236	4.9	v3.9 – v3.9	v3.8.13	1	CVE-2022-0494	4.4	v5.16.13 – v5.16.20	v5.16.12	8
CVE-2014-0205	6.9	v2.6.36.1 – v2.6.36.4	v2.6.36	4	CVE-2022-1011	7.8	v5.16.15 – v5.16.20	v5.16.14	6
CVE-2015-1593	5.0	v3.18.9 – v3.18.9	v3.18.8	1	CVE-2022-1055	7.8	v5.16.6 – v5.16.20	v5.16.5	15
CVE-2015-4170	4.7	v3.12.7 – v3.13.3	v3.12.6	72	CVE-2022-1198	5.5	v5.16.15 – v5.16.20	v5.16.14	6
CVE-2015-8944	5.5	v4.6 – v4.7	v4.5.7	9	CVE-2022-1263	5.5	v5.17.3 – v5.17.15	v5.17.2	13
CVE-2016-10906	7.0	v4.4.191 – v4.4.302	v4.4.190	112	CVE-2022-1353	7.1	v5.16.19 – v5.16.20	v5.16.18	2
CVE-2016-2085	5.5	v4.4.2 – v4.4.8	v4.4.1	7	CVE-2022-1419	7.8	v5.5.5 – v5.5.19	v5.5.4	15
CVE-2016-2384	4.6	v4.4.2 – v4.4.8	v4.4.1	7	CVE-2022-1734	7.0	v5.17.7 – v5.17.15	v5.17.6	9
CVE-2016-2550	5.5	v4.4.4 – v4.4.8	v4.4.3	5	CVE-2022-1852	5.5	v5.18.2 – v5.18.19	v5.18.1	18
CVE-2016-5400	4.3	v4.6.6 – v4.6.6	v4.6.5	1	CVE-2022-2078	5.5	v5.18.2 – v5.18.19	v5.18.1	18
CVE-2016-6156	5.1	v4.6.6 – v4.6.6	v4.6.5	1	CVE-2022-2318	5.5	v5.18.10 – v5.18.19	v5.18.9	10
CVE-2016-9604	4.4	v4.10.13 – v4.11	v4.10.12	6	CVE-2022-2380	5.5	v5.17.2 – v5.17.15	v5.17.1	14
CVE-2017-1000407	7.4	v4.14.6 – v4.14.325	v4.14.5	320	CVE-2022-2905	5.5	v5.19.6 – v5.19.17	v5.19.5	12
CVE-2017-12762	9.8	v4.12.5 – v4.12.5	v4.12.4	1	CVE-2022-3078	5.5	v5.17.2 – v5.17.15	v5.17.1	14
CVE-2017-16643	6.6	v4.13.11 – v4.13.11	v4.13.10	1	CVE-2022-3303	4.7	v5.19.9 – v5.19.17	v5.19.8	9
CVE-2017-18216	5.5	v4.14.57 – v4.14.325	v4.14.56	269	CVE-2022-3543	5.5	v6.0.3 – v6.0.19	v6.0.2	17
CVE-2017-18224	4.7	v4.14.57 – v4.14.325	v4.14.56	269	CVE-2022-3594	5.3	v6.0.3 – v6.0.19	v6.0.2	17
CVE-2017-5669	7.8	v4.10.2 – v4.10.17	v4.10.1	16	CVE-2022-3595	5.5	v6.0.16 – v6.0.19	v6.0.15	4
CVE-2017-5986	5.5	v4.9.11 – v4.9.11	v4.9.10	1	CVE-2022-3707	5.5	v6.0.19 – v6.0.19	v6.0.18	1
CVE-2017-7518	7.8	v4.11.8 – v4.11.12	v4.11.7	5	CVE-2022-45869	5.5	v6.0.11 – v6.0.19	v6.0.10	9
CVE-2017-7558	7.5	v4.12.14 – v4.13	v4.12.13	2	CVE-2022-48502	7.1	v6.1.40 – v6.1.54	v6.1.39	15
CVE-2018-1000028	7.4	v4.14.16 – v4.14.23, v4.15.1 – v4.15.7	v4.14.15	15	CVE-2023-0469	5.5	v6.0.11 – v6.0.19	v6.0.10	9
CVE-2018-10853	7.8	v4.17.2 – v4.17.19	v4.17.1	18	CVE-2023-0590	4.7	v6.0.6 – v6.0.19	v6.0.5	14
CVE-2018-1120	5.3	v4.16.10 – v4.16.18	v4.16.9	9	CVE-2023-0615	5.5	v6.0.7 – v6.1.54	v6.0.6	68
CVE-2018-18281	7.8	v4.18.16 – v4.18.20	v4.18.15	5	CVE-2023-1079	6.8	v6.2.3 – v6.2.16	v6.2.3	14
CVE-2019-12819	5.5	v4.20.17 – v4.20.17	v4.20.16	1	CVE-2023-1206	5.7	v6.4.8 – v6.4.16	v6.4.7	9
CVE-2019-14814	7.8	v5.2.17 – v5.2.21	v5.2.16	5	CVE-2023-1249	5.5	v5.17.2 – v5.17.15	v5.17.1	14
CVE-2019-18282	5.3	v5.3.10 – v5.3.10	v5.3.9	1	CVE-2023-1513	3.3	v6.1.13 – v6.1.54	v6.1.12	42
CVE-2019-19036	5.5	v5.3.4 – v5.3.12	v5.3.3	9	CVE-2023-1990	4.7	v6.2.8 – v6.2.16	v6.2.7	9
CVE-2019-3460	6.5	v5.0.6 – v5.1	v5.0.5	17	CVE-2023-1998	5.6	v6.2.3 – v6.2.16	v6.2.2	14
CVE-2019-3901	4.7	v4.5.6 – v4.7.10	v4.5.5	21	CVE-2023-2002	6.8	v6.3.1 – v6.3.13	v6.3	13
CVE-2020-10720	5.5	v5.1.7 – v5.1.21	v5.1.6	15	CVE-2023-2019	4.4	v5.19.2 – v5.19.17	v5.19.1	16
CVE-2020-14314	5.5	v5.8.4 – v5.8.9	v5.8.3	6	CVE-2023-2162	5.5	v6.1.11 – v6.1.54	v6.1.10	44
CVE-2020-14381	7.8	v5.5.12 – v5.5.19	v5.5.11	8	CVE-2023-2177	5.5	v5.18.16 – v5.18.19	v5.18.15	4
CVE-2020-15436	6.7	v5.7.6 – v5.7.19	v5.7.5	14	CVE-2023-2598	7.8	v6.3.2 – v6.3.6	v6.3.1	5
CVE-2020-15437	4.4	v5.7.11 – v5.7.19	v5.7.10	9	CVE-2023-2985	5.5	v6.2.3 – v6.2.16	v6.2.2	14
CVE-2020-25641	5.5	v5.8.8 – v5.8.13	v5.8.7	6	CVE-2023-3111	7.8	v5.19.4 – v5.19.17	v5.19.3	14
CVE-2020-25656	4.1	v5.9.5 – v5.9.16	v5.9.4	12	CVE-2023-3141	7.1	v6.3.4 – v6.3.6	v6.3.3	3
CVE-2020-35508	4.5	v5.9.7 – v5.11.22	v5.9.6	229	CVE-2023-3159	6.7	v5.17.7 – v5.17.15	v5.17.6	9
CVE-2021-20261	6.4	v4.4.262 – v4.4.302	v4.4.261	41	CVE-2023-3161	5.5	v6.1.11 – v6.1.54	v6.1.10	44
CVE-2021-20320	5.5	v5.14.7 – v5.14.21	v5.14.6	15	CVE-2023-32254	8.1	v6.3.2 – v6.3.13	v6.3.1	12
CVE-2021-20321	4.7	v5.14.12 – v5.14.21	v5.14.11	10	CVE-2023-32258	8.1	v6.3.2 – v6.3.9	v6.3.1	8
CVE-2021-20322	7.4	v5.14.4 – v5.14.21	v5.14.3	18	CVE-2023-3268	7.1	v6.3.2 – v6.3.13	v6.3.1	12
CVE-2021-22555	7.8	v5.11.15 – v5.11.22	v5.11.14	8	CVE-2023-3269	7.8	v6.4.1 – v6.4.16	v6.4	16
CVE-2021-31916	6.7	v5.11.11 – v5.11.22	v5.11.10	12	CVE-2023-3439	4.7	v5.17.6 – v5.17.15	v5.17.5	10
CVE-2021-33033	7.8	v5.11.7 – v5.11.13	v5.11.6	7	CVE-2023-3609	7.8	v6.3.9 – v6.3.13	v6.3.8	5
CVE-2021-3411	6.7	v5.9.15 – v5.9.16	v5.9.14	2	CVE-2023-3611	7.8	v6.4.5 – v6.4.16	v6.4.4	12
CVE-2021-3483	7.8	v5.11.12 – v5.11.22	v5.11.11	11	CVE-2023-3812	7.8	v6.0.8 – v6.0.19	v6.0.7	12
CVE-2021-3501	7.1	v5.11.16 – v5.11.22	v5.11.15	7	CVE-2023-3863	4.1	v6.4.4 – v6.4.16	v6.4.3	13
CVE-2021-3573	6.4	v5.12.10 – v5.12.19	v5.12.9	10	CVE-2023-4004	7.8	v6.4.7 – v6.4.16	v6.4.6	10
CVE-2021-3659	5.5	v5.11.14 – v5.11.22	v5.11.13	9	CVE-2023-4128	7.8	v6.4.10 – v6.4.16	v6.4.9	7
CVE-2021-3679	5.5	v5.13.6 – v5.13.19	v5.13.5	14	CVE-2023-4133	5.5	v6.2.13 – v6.2.16	v6.2.12	4
CVE-2021-3732	5.5	v5.13.11 – v5.13.19	v5.13.10	9	CVE-2023-4147	7.8	v6.4.8 – v6.4.16	v6.4.7	9
CVE-2021-3736	5.5	v5.14.6 – v5.14.20	v5.14.5	15	CVE-2023-4389	7.1	v5.17.4 – v5.18	v5.17.3	13
CVE-2021-3739	7.1	v5.14.1 – v5.14.20	v5.14	20	CVE-2023-4394	6.0	v5.19.6 – v5.19.17	v5.19.5	12