SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem

Xiaochen Zou	Yu Hao	Zheng Zhang	Juefei Pu	Weiteng Chen	Zhiyun Qian
UC Riverside	UC Riverside	UC Riverside	UC Riverside	Microsoft Research, Redmond	UC Riverside
xzou017@ucr.edu	yhao016@ucr.edu	zzhan173@ucr.edu	jpu007@ucr.edu	weitengchen@microsoft.com	zhiyunq@cs.ucr.edu

***Abstract*—Continuous fuzzing has become an integral part of the Linux kernel ecosystem, discovering thousands of bugs over the past few years. Interestingly, only a tiny fraction of them were turned into real-world exploits that target downstream distributions, *e.g.,* Ubuntu and Fedora. This contradicts the conclusions of existing exploitability assessment tools, which classify hundreds of those bugs as high-risk, implying a high likelihood of exploitability. Our study aims to understand the gap and bridge it. Through our investigation, we realize that the current exploitability assessment tools exclusively test bug exploitability on the upstream Linux, which is for development only; in fact, we find many of them fail to reproduce directly in downstreams. Through a large-scale measurement study of 230 bugs on 43 distros (8,032 bug/distro pairs), we find that each distro only reproduces 19.1% of bugs on average by running the upstream PoCs as root user, and 0.9% without root. Remarkably, both numbers can be significantly improved by 61% and 1300% times respectively through appropriate PoC adaptations, necessitated by environment differences. To this end, we developed SyzBridge, a fully automated system that adapts upstream PoCs to downstream kernels. We further integrate SyzBridge with SyzScope, a state-of-the-art exploitability assessment tool that can identify high-risk exploit primitives, e.g., control flow hijack. Our integrated pipeline successfully identified 53 bugs originated from syzbot that are likely exploitable on downstream distributions, surpassing the mere 5 bugs that were turned into real-world exploits among 5,000 upstream bugs from syzbot. Notably, to validate the results, we successfully exploited 5 additional bugs that were previously not known to be exploitable publicly.**

I. Introduction

Linux kernel security has always been a popular area of great interest for security researchers. As an alternative way to help improve kernel security, exploiting Linux kernel is encouraged by influential organizations such as Google [10] and ZDI [26]. Numerous studies have specifically targeted the exploitability of the kernel, introducing innovative techniques for kernel exploits [53], expanding the attack surface [34], and enhancing the stability of exploits [68]. However, not every bug is exploitable. It is well-known that bugs that have high-risk primitives are likely to be exploited [70], *e.g.,* use-after-free write, double-free.

In recent years, the use of continuous fuzzing, *e.g.,* syzbot [40], has exposed thousands of Linux kernel bugs to the public. With the improvements of kernel fuzzing techniques [64], [31], [41], [69], [35], [63], even more bugs may be exposed in the future. Considering the enormous volume of kernel bugs, conducting manual inspections to assess the exploitability of every single one is unrealistic. This led to the development of bug assessment techniques [70], [52], [32], [66], [45], which attempts to classify bugs based on some notion of "exploitability", *e.g.,* uncovering high-risk exploit primitives. The automated exploitability assessments have successfully identified many exploitable bugs. Notably, SyzScope [70] has classified 183 bugs out of 1,170 fuzzer-exposed bugs as high-risk. KOOBE [32] has managed to generate 6 new exploits for previously non-exploitable bugs. Despite these promising outcomes, the occurrence of real-world Linux kernel exploits remains surprisingly rare. Indeed, syzbot [40] has reported only 5 exploitable kernel vulnerabilities originating from upstream kernel bugs since 2017. However, our subsequent evaluation revealed that we discovered over 50 exploitable kernel vulnerabilities across distros such as Fedora, Ubuntu, Debian, and SUSE. This discrepancy prompts us to investigate the missing step that creates a gap between the relatively low number of real-world exploits and the promising results produced by assessment tools.

We found that prior solutions assessed exploitability against the upstream kernel, i.e., Linux mainline, only, without verifying whether the results will transfer to the downstream kernels [70], [32]. In practice, the upstream kernel is used for development only and it is the downstream Linux distros that are used in the real world. In other words, prior solutions implicitly assumed PoCs generated by fuzzing upstream kernels seamlessly apply to downstream distros. Our study questions the assumption.

In our large-scale measurement study at §V-B, we examined 230 upstream kernel bugs across 42 downstream distros. The findings revealed that a significant majority of the upstream kernel bugs cannot even be reproduced when subjected to exploitability assessment tools in real-world downstream distributions. Remarkably, out of the 183 high-risk bugs identified by SyzScope, only 39 of them were triggerable on at least one of the following Linux distributions: Ubuntu, Fedora, Debian, and SUSE. Moreover, only 2 bugs could be triggered by a normal user, which is a necessary condition for exploitable bugs. These findings demonstrate that existing

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24926
www.ndss-symposium.org

exploitability assessment tools have limited real-world impact, thereby explaining why they haven't significantly increased the number of real-world exploits.

We hypothesize that it will likely not work by simply re-executing the upstream PoC against a downstream, due to the differences between production kernels and development kernels. In this paper, we are motivated to answer two questions: **(1) why an upstream PoC would fail against a downstream kernel and how prevalent it occurs** and **(2) whether the bug can be reproduced without requiring high privileges.**

To answer the two questions, we perform a comprehensive analysis on a smaller-scale dataset, aiming to gain insights into the underlying factors that lead to unsuccessful bug reproduction on downstream distros. Initially, we hypothesized that the primary reason for these failures would be discrepancies in the kernel code context between upstream and downstream. For instance, this could lead to deviations in the execution trace, as observed in certain user-space software [36]. However, following our investigation, we were surprised to discover that this was only a rare occurrence. In reality, the failure to reproduce these bugs stems from various factors, which we collectively refer to as *environment differences*.

Intriguingly, based on our analysis, a significant portion of these differences can be mitigated by making adjustments to the PoC. However, trivial solutions like blind fuzzing/mutating are insufficient for completing such adjustments. For example, one major reason for this failure is the absence of critical kernel modules, which are compiled but not loaded. The common approach of using `modprobe` to load a module requires root privileges, making it impractical to be exploited. Additionally, a typical downstream distro kernel like Ubuntu consists of approximately 6,000 modules, making it impossible to blindly test and determine the missing ones. To address this challenge, we employ a unique approach. It extracts PoC execution traces from upstream and downstream, carefully identifies their divergences, and pinpoints the missing kernel modules. Furthermore, we utilize static analysis and fuzzing techniques to discover a special kernel internal module loading mechanism. This mechanism is then leveraged to load the target missing module without requiring root privileges.

Therefore, to better assist and integrate with conventional exploitability assessment tools, we further develop SyzBridge, an automated system that diagnoses the failures of a PoC on a given downstream distro and attempts to make appropriate changes to the PoC. The objective is twofold: (1) to successfully reproduce the bug in a downstream kernel, and (2) to reduce the privilege requirement. We envision that SyzBridge would not just contribute to solving the bug triggerability issues, but also bring another dimension to the existing tools, making the exploitability assessment results have more real-world impacts.

The evaluation results are promising. Using SyzBridge on the same dataset of 230 upstream bugs (8,032 bug/distro pairs), we improved the number of triggerable bugs by 61% and the number of normal-user-triggerable bugs by 1300%. To demonstrate the real-world impact of our results, SyzBridge integrates with SyzScope – a state-of-the-art exploitability assessment tool, in the pipeline. The integrated pipeline analyzed 282 upstream high-risk bugs and revealed 53 bugs that are likely exploitable by normal users on some downstream distros, compared to just 5 that were considered as such (with previously-assigned CVEs). We sampled additional 5 bugs from the 53 likely exploitable bugs to develop kernel exploits. Those exploits demonstrate the efficiency of the pipeline in exploitability assessment. Notably, one of the exploits is an end-to-end exploit against the latest Ubuntu distro, which resulted in one additional CVE. In summary, this paper makes the following contributions:

- We introduced a novel and previously-neglected exploitability assessment dimension to the Linux kernel ecosystem. The dimension addresses downstream bug triggerability issues and equips existing exploitability tools with a new capability to work on real-world downstream distros.
- We developed SyzBridge, an automated system designed to enhance the chances of reproducing bugs on downstream distros by making adaptations to the original upstream PoC. Our system successfully reproduced 61% more bugs, many of them don't require root privileges. To facilitate tool integration and future research, we released our source code at https://github.com/seclab-ucr/SyzBridge
- Our system has been designed to facilitate integration into the existing exploitability assessment tool pipe. The pipeline reveals 53 likely exploitable bugs on downstream kernels, compared to only 5 that were previously known.

II. Background and Overview

A. Linux Ecosystem

As an open-source operating system, Linux kernel is widely reused and customized, forming an ecosystem with multiple concurrent branches under development and maintenance. Linux mainline is a development-only branch that is considered the upstream where both new features and bug fixes occur. Whenever the Linux mainline reaches a milestone, *e.g.,* major update has been published, it is forked into either a stable or a long-term-support (LTS) branch. In such branches, developers focus on improving stability and fixing bugs without incorporating new features. The stable and LTS branches are then relied upon by downstream kernels which are actually used in production (also called Linux distributions or Linux distros), *e.g.,* Ubuntu [22], Debian [4], Fedora [6], Suse [17].

Downstream kernels diverge from the mainline and even stable/LTS branches for several reasons. It is obvious that a downstream kernel that follows stable/LTS for its entire lifetime will be different from mainline, increasingly so as more and more features go into mainline (*e.g.,* Debian and Ubuntu). Furthermore, a downstream kernel will customize the kernel on top of the stable/LTS for a variety of reasons [23], [5]: supporting specific user-space features (*e.g.,* Debian packaging) and out-of-tree drivers, enabling or disabling certain kernel features or modules (via kernel configs), cherry-picking important patches (*e.g.,* security) from mainline that are not (yet) in stable/LTS.

B. Syzbot

As mentioned, syzbot is a continuous kernel fuzzing platform operated by Google, where bugs are discovered and reported on a public dashboard [40]. It targets primarily

Linux mainline with most of its fuzzing instances. It also has limited fuzzing instances on several old LTS branches, *e.g.,* v4.14.y [19]. For every newly discovered bug, syzbot aims to generate PoCs that assist developers in reproducing the same bug for testing purposes.

The way syzbot operates is that it compiles the latest kernel (mainline or LTS) from the public git repository on a daily basis and fuzzes it for roughly a day, and repeats. In other words, when fuzzing mainline, it can effectively find bugs as they are introduced during the development process – this is why it dedicates most of the fuzzing resources to mainline. Moreover, to achieve high coverage, when compiling the kernel, syzbot enables the most commonly used options (*e.g.,* translating into kernel features) in the config file as subject to certain conflicts [18]. It also enables a number of sanitizer-related options [27], [39], [28] to instrument the kernel and facilitate bug detection at runtime. Finally, syzbot always fuzzes the kernel with test cases that run as root user, allowing the fuzzer to discover bugs that would require privileges to trigger.

C. Bug exploitability

Bug exploitability is contingent upon certain high-risk primitives. To the best of our knowledge, exploitable Linux kernel vulnerabilities require at least one high-risk primitive. As noted by SyzScope [70], high-risk primitives include various memory write primitives, *e.g.,* use-after-free write and out-of-bounds write, as well as memory free primitives, *e.g.,* arbitrary free and double free. Additionally, control flow hijacking is also a well-known high-risk primitive.

Exploitability assessment tools like KOOBE [32] rely on discovered high-risk primitives (e.g., out-of-bounds write) to overwrite critical kernel data, thereby generating concrete exploits. SyzScope assesses bug exploitability by elevating low-risk primitives (e.g., memory read and non-security primitives) to high-risk primitives. Both tools have achieved noteworthy results. For instance, KOOBE successfully generated 6 new exploits from 4 Linux upstream bugs, while SyzScope elevated 183 out of 1,173 upstream low-risk bugs to high-risk bugs.

Despite the notable success of exploitability assessment tools, the impact on the number of real-world exploits appears to be limited. As mentioned previously, these tools primarily focus on the upstream development kernel and rely on original upstream PoCs without any adaptation. In fact, a lot of downstream exploitable bugs cannot be directly triggered by the upstream PoC, thereby neglecting their real-world impacts on downstream distribution kernels. We will showcase in the next section.

D. Motivating Example

In order to highlight the critical gap in existing exploitability assessment tools, we present a real bug [9] from syzbot as an illustrative example. This bug exhibits a high-risk out-of-bounds write as an initial primitive, making it likely to be transformed into a real-world vulnerability.

However, the current exploitability assessment tools are incapable of analyzing this bug because it does not reproduce on the target Ubuntu kernel even when running the original PoC with root privileges. Fortunately, SyzBridge managed to diagnose the root cause and generate a new PoC specifically tailored for the Ubuntu kernel. Without SyzBridge, nobody realizes the potential exploitability of this upstream bug, at least according to publicly available information. We developed an end-to-end exploit for this 1-day bug in the latest Ubuntu kernel at the time and received credit from the community. To address ethical concerns, we provide a detailed account of how we handled this exploit and responsibly reported it to the developers in §VI-D.

```
1   void km_state_notify(struct xfrm_state *x, const struct
        km_event *c)
2   {
3           struct xfrm_mgr *km;
4           rcu_read_lock();
5           list_for_each_entry_rcu(km, &xfrm_km_list, list)
                  // Getting km from the global list xfrm_km_list
6                   if (km->notify)
7                           km->notify(x, c);
8           rcu_read_unlock();
9   }
10
11  void xfrm_register_km(struct xfrm_mgr *km)
12  {
13          spin_lock_bh(&xfrm_km_lock);
14          list_add_tail_rcu(&km->list, &xfrm_km_list);
15          spin_unlock_bh(&xfrm_km_lock);
16  }
17
18  #define list_for_each_entry_rcu(...) \
19          for (__list_check_rcu(dummy, ## cond, 0), ...) \
```

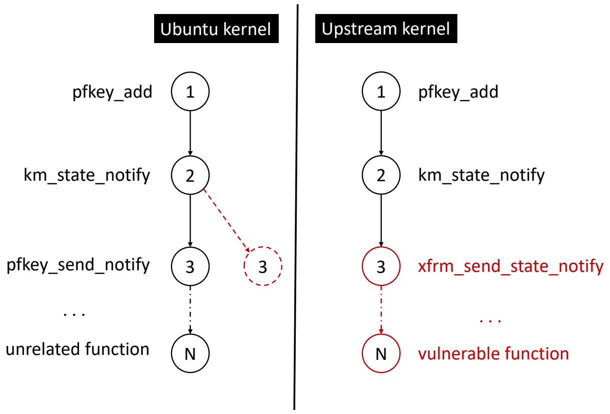


Figure 1: Motivation example

This vulnerability occurs in kernel netfilter. `km_state_notify` in Figure 1 is the critical function that invokes the corresponding netfilter. This invocation occurs at line 7 through the xfrm manager `km`. The value of `km` is assigned at line 5 from a global linked list, `xfrm_km_list`. Note that `list_for_each_entry_rcu` iterates the global linked list, `xfrm_km_list`, and invokes the correct `notify()` function. However, this global linked list must be initialized before being used by `km_state_notify`. The initialization process is depicted at line 14 within the `xfr_register_km` function. In the upstream kernel, this initialization function is automatically called during the kernel booting process when the `xfrm_user` module is loaded. On the other hand, the default configuration of the Ubuntu kernel does not load the `xfrm_user` module, resulting in the initialization function not being completed, and eventually causing the bug cannot be reproduced.

As shown in Figure 1, `km_state_notify()` invokes an indirect call at line 7. If we compare the trace from Ubuntu with the trace from upstream kernel, we note that the callees are distinct. This is because Ubuntu kernel only has the default modules `pfkey` for `notify` at this moment, which causes it to call into the common function `pfkey_send_notify()`. SyzBridge extracts the traces and automatically locates the unmatched trace node (node 3) and reveals the missing function `xfrm_send_state_notify()` (the technical details will be discussed in §IV-C). SyzBridge locates the corresponding module to be `xfrm_user`, and loads it through `modprobe` to trigger the bug on Ubuntu, finally the PoC successfully triggers the bug.

However, the standard method of loading modules through `modprobe` requires root privileges, making it impractical for exploiting a vulnerability. SyzBridge uses a unique technique that leverages a kernel internal mechanism to seamlessly load the target module from an unprivileged user (more details will be discussed in §IV-D). In addition to downgrading the privilege for module loading, SyzBridge also automatically downgrades the privilege of using network modules by identifying kernel security checks and applying user namespaces (more details will be discussed in §IV-E). As a result, the vulnerability can now be successfully triggered on Ubuntu with normal privilege

Moreover, SyzBridge seamlessly integrated SyzScope, establishing a comprehensive pipeline for assessing bug exploitability on downstream distro. By examining the analysis results obtained from this pipeline, we identified several of the most useful capabilities and utilized them to construct the final exploit.

Table I: Distro Dataset for Manual Investigation

	Distro Major Release	Code Name	Kernel Version	Released Date
Ubuntu	Ubuntu-20.04	focal	5.4.26	Apr 23 2020
	Ubuntu-20.04.1	focal	5.4.42	Aug 6 2020
	Ubuntu-20.04.2	focal	5.4.65	Feb 4 2021
	Ubuntu-20.04.3	focal	5.4.81	Aug 26 2021
Fedora	Fedora-32	32	5.6.6	Apr 28 2020
	Fedora-33	33	5.8.16	Oct 27 2020
	Fedora-34	34	5.11.12	Apr 27 2021
	Fedora-35	35	5.14.10	Nov 2 2021
Debian	Debian-10.4	buster	4.19.118	May 9 2020
	Debian-10.5	buster	4.19.132	Aug 1 2020
	Debian-10.6	buster	4.19.146	Sep 29 2020
	Debian-10.7	buster	4.19.160	Dec 5 2020
	Debian-10.8	buster	4.19.171	Feb 6 2021
	Debian-10.9	buster	4.19.181	Mar 27 2021
	Debian-10.10	buster	4.19.194	Jun 19 2021
	Debian-10.11	buster	4.19.208	Oct 9 2021
Suse	SLE-15-SP2	15	5.3.18-22	Jul 21 2020
	SLE-15-SP3	15	5.3.18	Jun 22 2021

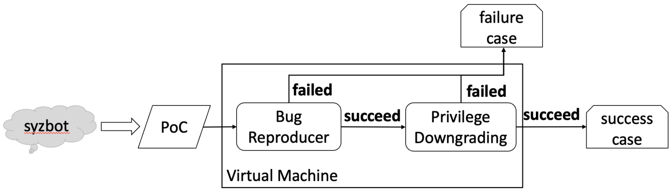


Figure 2: Exploratory Experiment Workflow

III. Exploratory Experiment

In order to understand why upstream bug PoCs cannot reproduce the same bug on downstream, we set up an exploratory experiment.

A. Dataset & Experiment Setup

Syzbot bugs. We choose all KASAN bugs that were found against Linux upstream on syzbot and had C PoCs within the time frame of Jan 1st 2020 to Dec 31th 2022. This leads to 225 bugs and corresponding C PoCs. To complement the dataset, we also find 5 CVEs that originated from syzbot in the same time frame that had been proven exploitable according to the public response of kCTF [10], [13]. We realize that 3 of the CVEs are already covered in the upstream KASAN bug; interestingly, the other 2 are also KASAN bugs but are reported in other kernel targets (*e.g.,* long-term-support), which had C PoCs as well. In total, we have 230 bugs from syzbot.

Downstream distros. We use the four popular distributions [12] as our distro dataset: Ubuntu, Fedora, Debian, and Suse. We gathered their major releases that cover Jan 1st, 2020 to Nov, 25th 2022, as shown in Table VII. For each syzbot bug, we pick the potentially affected major release kernels for testing, *i.e.,* those that satisfy the two requirements: (1) The bug has not been patched in the downstream kernel. (2) The major release's corresponding branch is still under support at the time when the bug is found (see Table VII for details). Please note that even though some major releases are now out of support, they were still under support when the bug was discovered (e.g., a Jun 2020 bug on Ubuntu-20.04). In other words, it includes a retrospective analysis of past data. In total, we gathered 43 distros and made 8,032 bug/distro pairs.

Experiment setup. In order to capture `KASAN` bugs (all of the selected bugs are found by `KASAN` detectors), we enabled `CONFIG_KASAN` in the distro kernel configuration. In addition, we enable a few other kernel debug features such as `CONFIG_FAULT_INJECTION`, `CONFIG_DEBUG_KERNEL`, *etc.*, as they are also used by syzbot. Besides them, we keep the remaining options the same as in the original configuration to make the compiled kernel as close to the end-user version as possible.

As shown in Figure 2, we launch a separate virtual machine for each valid distro version for a given bug/PoC. Initially, we run the PoC as the root user inside the VM. If the bug is triggered successfully, we then run the PoC again as a default and unprivileged user if the first step succeeds (triggered the bug), and record any failures during either step. We give 600 seconds for each PoC to run. We ran SyzBridge on Ubuntu-18.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @ 2.50GHz * 2.

B. Exploratory Experiment Results

After running 230 PoCs against 43 distros, we discovered only a fraction of bugs are triggerable by the root user across all the distros — 55, 58, 33, and 29 out of 230 total for Ubuntu, Fedora, Debian and Suse respectively. Furthermore, only 2, 3, 2, and 1 (0.4% - 1.3%) are triggered directly by an unprivileged user. Among 5 CVEs that originated from syzbot, 4 of them were triggered by root, and none of them were triggered by an unprivileged user. Note that we consider a bug to be successfully triggered on a distro vendor as long as it is triggered on one major release from the vendor.

The exploratory experiment results indicate only a small portion (19.1% on average across the four distros) of upstream bugs actually affect downstream distros. Surprisingly, there is an extremely low percentage of bugs (0.9% on average across the four distros) that are triggerable by an unprivileged user. It suggests that most fuzzer-exposed bugs do not have a serious impact on downstream kernels. To determine whether the results represent the truth, we sampled some of the failed cases to conduct a manual root cause analysis.

C. Manual Investigation Dataset & Setup

We randomly picked 50 `KASAN` bugs that failed to reproduce on target distros or only reproducible by root users. We limit the bugs to the time frame from 2020 to 2021 according to their first discovery date on syzbot [40]; this is because they are more likely to be patched and can ease the manual investigation. We then pick the corresponding major releases of the four distros that were released within that time frame. The details are shown in Table I.

For each bug, we limit the testing for each vendor to a single kernel release — the most recent release right before the bug was found. For example, if a bug was reported on July 1st 2021, then Ubuntu-20.04.2, Fedora-34, Debian-10.10, and SLE-15-SP3 from Table I are chosen for testing. In total, this leads to 200 bug/distro pairs.

We first run the PoC as root user on the target distro. If the bug fails to reproduce on a target distro, we manually analyze the root cause of this failure. For those bugs that can reproduce with root, we rerun the PoC as an unprivileged user to observe the triggerability again. If they cannot trigger the bug by an unprivileged user, we start another manual analysis to understand the privilege requirement.

D. Manual Investigation Results

Among these 200 bug/distro pairs, 120 of them do not affect the target distro. This is because the buggy commits, *i.e.,* the commits that introduce the bugs (as given in the `Fixes` tag of a patch [16]), simply do not exist in the downstream kernel. For the remaining 80 pairs where the downstream kernels do contain the buggy code, we list the results in Table II. By simply running the original upstream PoC, we observe only 18 out of 80 bug/distro pairs were successful across all four vendors, *i.e.,* the PoC manages to trigger the bug as root. In contrast, none of them are triggered by unprivileged users. We then conduct a manual analysis to understand the root causes of (1) the remaining 62 pairs where the PoCs failed to trigger as root, and (2) the 18 pairs that did trigger successfully with root but not with unprivileged users. The results are summarized below.

1) Necessary logic missing: We found 41 out of 62 pairs failed because the necessary logic (*e.g.,* functions) is missing. For example, the buggy commit exists but the function involved in the buggy commit is not compiled in the downstream kernel. Note that this is different from the case where the buggy commit does not exist (which we already filtered earlier). However, the implication is the same — there is no adaptation possible to the PoC to trigger the bug.

2) Code context change: We noticed 1 out of 62 pairs failed because of the downstream kernel code context change. Specifically, the downstream kernel had a check that was not present in the upstream such that the PoC needs to be adapted slightly (changing a constant) to trigger the bug. In this paper, we deliberately exclude such cases from our scope because: (1) there is only a limited number of such cases (also supported in our larger-scale experiment later). (2) such code context change can be addressed by prior solutions using fine-grained program analysis techniques [45], [36].

3) Environment Requirements Unsatisfied: In the remaining 20 out of 62 pairs, we found the root causes are interestingly not related to the core logic of the bug (*e.g.,* code context changes). Instead, they are due to what we consider *environment differences* in the downstream OS. In general, we categorize them into three types of environment differences. Note that a PoC can fail due to more than one reason.

R1: Preparation steps failing. We discovered 3 pairs failed due to the lack of debugging devices in target distros. For example, `/dev/raw-gadget` [24] is a low-level interface for the Linux USB Gadget subsystem. In syzbot, such preparation steps always occur in a specially marked part of the PoC (at the beginning) and therefore easy to recognize. It is not supposed to be included in a production distro kernel and indeed we do not enable it when we compile the downstream kernels. When a PoC tries to open this device on the downstream OS, it always fails and terminates. This is not a problem for the upstream kernel because such debugging features are always enabled in the kernel config.

R2: Distro background noises. There are 13 pairs that failed due to distro background noises. These noises come from the daemon processes that are running all the time in the background, *e.g.,* system services. This is expected because these downstream OSes are designed to be functional. In contrast, the upstream Linux that syzbot uses is minimally functional with few default processes that are long-running. These background processes cause two types of issues preventing the bug from being triggered.

- Distro resources needed by PoCs are occupied by other processes. For example, `loop` devices [11] provide a block device for a filesystem image to be mounted (*e.g.,* .iso image). We observe that `loop` devices on distro images are often occupied by other processes.

- Race condition failed due to noises. With minimal background processes, an upstream kernel can easily trigger a race condition bug, sometimes in a single execution (without looping at all). This result tricks the fuzzer into thinking that the bug is not really a race condition bug, so the fuzzer failed

Table II: Breakdown of investigation results

Distro	Triggered by		R1	R2	R3	R4
	root	normal				
Ubuntu	5	0	2	4	4	2
Fedora	5	0	0	2	2	4
Debian	3	0	1	4	2	0
Suse	5	0	0	3	2	2

to mark it as such and will not repeat the race in the PoC. However, the same PoC faces much more background noise from long-running processes in downstream OSes and a single race almost always fails.

R3: Necessary kernel modules not loaded. We noticed 10 pairs failed because the bug required specific modules that are compiled and included in the kernel, but not loaded by default. Note that this is different from the previous reason "Necessary logic is missing" where the necessary code is simply not present or compiled, causing no chance of triggering the bug. In contrast, if the modules are included in the kernel but are not loaded yet, there is technically still a chance to trigger the bug by loading the modules. In other words, the true attack surface of downstream kernels needs to account for the modules that are loadable on demand, especially those that can be loaded by an unprivileged user (which we discuss in §IV-D).

4) Privilege Requirement: In order to understand how privilege plays a role in the bug-triggering process, we investigate the root causes that prevent a non-privileged user from triggering the bug successfully.

R4: Kernel security checks. In Linux kernel, each process has its own `cred` struct representing its privileges, and the security checks are all related to the fields defined in this structure. Generally, there are two common types of checks: *uid and gid checks*. A `uid` and `gid` represent user id and group id the process is running as (there are also `euid` and `egid`, *etc.* and we refer to them generally as `uid` and `gid`). The highest privilege levels for `uid` and `gid` are called root (constants of zero) in Linux.

The other type of check is *capability checks*. Capabilities are finer-grained compared to `uid` and `gid` where privileges are divided into smaller units that can be enabled or disabled independently for each process, *e.g.,* `CAP_NET_ADMIN` giving a process the privilege to use raw sockets. Figure 3 from case study is showing a case that `CAP_NET_ADMIN` check fails because the calling process does not have the corresponding capability.

When syzbot fuzzes the upstream kernel, the PoC process runs as root and is granted every capability and therefore has no problem passing these checks. However, running such a PoC with an unprivileged user on downstream OSes will fail to trigger the bug because of these checks.

IV. SyzBridge

In order to facilitate exploitability assessment on downstream kernels, it is crucial to have a system that not only addresses individual triggerability issues but also possesses the capability to seamlessly integrate with different exploitability assessment tools. With this in mind, we have designed and implemented SyzBridge, an end-to-end solution that takes an upstream PoC and automatically assesses its triggerability in downstream kernels. SyzBridge encompasses a range of fully automated workflows, including kernel deployment, bug reproduction, and crash detection.

The core functionality of SyzBridge is implemented through approximately 8,000 lines of Python code, comprising all the essential components required for bug assessment and subsequent analysis. It consists of an upstream bug crawler, virtual machine management, Proof of Concept (PoC) handlers, and various plugins (for adaptations).

To address failure reasons R1 to R4, we have devised and implemented individual plugins, totaling approximately 4,000 lines of code, to adapt to each specific failure reason. These plugins ensure that SyzBridge effectively handles the identified challenges. We describe them in detail below.

A. Environment Adaptation for Failed Preparation Steps.

As mentioned in R1, syzbot PoCs encode a variety of preparation steps. This is because different kernel functionalities may require additional setup processes. To correctly fuzz a particular kernel functionality (*e.g.,* USB, WiFi), syzkaller needs to prepare the fuzzing environment first, and some of the preparations require additional interfaces that the production kernels do not have. For example, to effectively fuzz kernel USB functionality, the syzbot kernels are compiled with the interface `/dev/raw-gadget` exposed to userspace, so that it is possible to provide simulated input from a fake USB device. Of course, not every PoC actually requires such an environment, and it is highly dependent on whether the subsequent syscalls (beyond the preparation steps) in fact interact with the related functionalities. Note that when syzkaller tries to generate a PoC, it has a procedure to minimize the test case. However, it appears to be insufficient in minimizing the preparation steps. As a result, the unnecessary preparation steps would be kept in the final PoC, which would fail on a downstream OS that does not have such debugging interfaces. To adapt such PoCs on a downstream OS, we perform our own minimization on the preparation steps. In short, we iteratively disable existing preparation steps and find the minimal steps that are actually necessary for the bug triggering.

B. Environment Adaptation for Background Noise

As mentioned in R2, background noises often come from daemon processes and services. These processes are long-running in the background, and constantly compete for resources with other programs (*e.g.,* the PoC). Unfortunately, some daemon processes and services even have higher priority — they execute earlier during the boot process. As a result, it is difficult, if not impossible, for the user programs to win the resources. If these pre-occupied resources are essential to the PoC, the bug would not be triggered. For the purpose of testing bug triggerability, we simply adapt the PoC to force the kernel to give up the resources we want from other processes. For example, a `loop` device is essential for mounting a file system before bug triggering. But most `loop` devices are occupied by background processes on many distro OSes. If we find the designated resources are all busy, we force the kernel to release them (*e.g.,* unmount) and then continue our PoC.

Regarding the race condition issue, as we mentioned, a bug is fundamentally a race condition bug but syzbot failed to recognize it and incorrectly labeled the PoC as non-repeat, because of less background noise in the upstream Linux used by syzbot. To mitigate this problem, we treat every bug as a potential race condition bug. If a PoC is running as a single process, we launch multiple PoC processes (*i.e.,* 6, the number of processes that syzbot usually use for race condition bugs) to make them collide with each other. Furthermore, we adapt the PoCs to always run in a loop increasing the chance of winning the race (with a timeout of 600 seconds).

C. Environment Adaptation for Module Loading

As mentioned in R3, due to the fact that downstream kernels are often compiled with many individual loadable kernel modules, they have to be loaded before the PoC can run. In theory, as root user, we can simply forcefully load all available kernel modules in downstream as an adaptation. However, as we will discuss later in §IV-D, it is beneficial to pinpoint the exact module(s) so we can later try to load these modules without being root. And there are thousands of kernel modules in downstream kernels, which makes it infeasible to test every single possible combination.

1) Pinpointing explicitly dependent but missing modules: We denote the "explicitly dependent but missing modules" in downstream kernels as Dep_{exp}. They represent modules that are exercised by a PoC when executed in the upstream kernel, but are not loaded in downstream.

We first collect the complete function-level kernel execution trace (via `ftrace` [8]) of the PoC when executed in both the upstream OS (reference trace) and the downstream OS (target trace). Each trace includes all the syscalls and the corresponding internal kernel functions. We then look up the kernel module to which each internal kernel function belongs. Specifically, we look up the module name by the corresponding source file name from the Makefile [48].

Our next step is to check which of these modules can be loaded in the downstream. At this point, we check each module in the set in terms of its compilation status. There are three possibilities: 1. *The module is not compiled.* 2. *The module is compiled into the kernel core (built-in module).* 3. *The module is compiled as a loadable add-on (loadable module).*

In the first two cases, we will discard the module from Dep_{exp} because there is either no way or no need to load the module. Only in the third case, we will retain the module, subject to further minimization (as will be discussed in §IV-C3).

2) Pinpointing implicitly dependent but missing modules: We denote the "implicitly dependent but missing modules" in downstream kernels as Dep_{imp}. They represent the modules whose code did not show up in the execution trace but the absence of them prevents the bug from triggering.

In the case study (Figure 3), we will show a case where the global `net_device` linked list was queried by the execution trace, but the code that inserts the object to the linked list is conducted by the module initialization logic in a separate process. For instance, the kernel function `register_netdevice` is called to insert a network device into the global `net_device` linked list during device initialization. The order of insertion does not matter because each network device in the list is assigned a unique identifier, `ifindex`. This unique identifier can be used to retrieve a specific device. The size of the device linked list varies among different distros but typically remains relatively small, usually no more than a few dozen. By observing the execution trace, we can only collect Dep_{exp} but not Dep_{imp}.

Our solution is to conduct a static analysis leveraging a recent tool specifically designed for dependency analysis of the kernel [42]. The existing analysis was originally designed to identify, *within a kernel module*, how a piece of global memory (*e.g.,* global variable) read in one syscall is written in another. However, in our case, we are only interested in such dependencies *across kernel modules*. Nevertheless, the methodology is similar. All we need to do is to identify global variables that are read in one kernel module and written in another. In addition to constraining the tool to "cross-module" dependencies, we also constrain the modules that perform write operations on the global variable to those that occur in `module_init` functions. In other words, the write has to happen as soon as the module is loaded. The constraint is imposed because of the following reason: if a write operation has to be triggered through additional syscalls, *e.g.,* `ioctl()`, it must have been included in the original upstream PoC, rendering the module that contained the write operation an explicitly dependent module (instead of an implicitly dependent one).

We conducted the dependency analysis on the upstream kernel with `allyesconfig` to include as many kernel modules as possible. We then performed a final step of offline manual analysis to determine what global variables are shared across modules and which ones can cause a "read" module to abort if the "write" module is not loaded. We finally identified three different global linked lists that are most commonly shared across modules, representing character, block, and network device drivers. We therefore encode the knowledge regarding their operations. Specifically, when we observe the execution trace in the downstream kernel that performs any "read" operation (*e.g.,* linked list lookup), we will add a corresponding "write" module that is compiled as a loadable module into Dep_{imp}.

3) Module minimization: Now that we have both Dep_{exp} and Dep_{imp}, we will then decide which ones are truly necessary for the bug to trigger. The reason is that these "dependent" modules are not necessarily "required". For instance, there are conditionally compiled code marked by `#ifdef` macros which are enabled in upstream kernels but not downstream ones. Such code may call into additional modules but is not required for the bug to trigger. Implicitly dependent modules may also not be "required" as the "read" module may not be critical, *e.g.,* when looking up the linked list fails, it may revert to an alternative path and continue to trigger the bug.

Note that syzkaller already performed its own test case minimization for PoCs after the bug was triggered. However, syscall minimization is still different from module minimization. This is because syzkaller's minimization operates at the syscall level — it is possible that a syscall is absolutely necessary to trigger a bug, but whether the syscall exercised another module (through callbacks) is not of its concern. The built-in minimization in syzkaller pinpoints the guilty syscalls and rules out the unnecessary ones so that developers can better understand the bug. Similarly, the minimization of required modules helps us understand the requirement of the bug, and its potential impact on downstream kernels. Yet, being able to minimize the "required" modules for a PoC will improve the odds of triggering the bug without the root privilege (see §IV-D), as many modules do require it to load.

We implement the module minimization procedure in a similar fashion to syscall minimization [21], except that we add modules one by one as opposed to removing syscalls one by one. We chose this because we find that in most cases the number of "required" modules is actually much smaller than the total number of modules in Dep_{exp} and Dep_{imp}.

D. Privilege Adaptation for Module Loading

As alluded to earlier, modules can require privileges to load. A universal method to load a kernel module is through the `modprobe` utility program, which invokes the `init_module()` syscall. Unfortunately, the `init_module()` function checks user's initial namespace `init_user_ns` for a special capability called `CAP_SYS_MODULE`. This capability is not present in an unprivileged user's initial namespace. Only the root user or a user who has been granted this capability can execute the `init_module()` syscall. If a bug requires additional modules, loading them through `modprobe` necessitates the root privilege requirement. However, Linux kernel also provides an inconspicuous way to load modules through kernel syscalls by unprivileged users [14]. These modules can be loaded automatically when certain syscalls such as `socket()` are invoked. The `xfrm_user` module is not loaded in Ubuntu by default, but can be loaded by an unprivileged user through `syscall(__NR_socket, 16, 3, 6);`.

In order to discover all such kernel modules systematically, we identify a key internal kernel function named `request_module()` that by itself does not have privilege requirements. We find that there are hundreds of invocations of `request_module()` scattered in the kernel. It is not possible to manually analyze all invocations and construct concrete test cases that reach `request_module()`. Therefore, we developed a guided fuzzer on top of syzkaller to generate test cases automatically that can load modules by unprivileged users.

Our basic idea is to constrain the syscall search space to those that have a chance to reach `request_module()`. Naturally, we need a call graph analysis, which should resolve indirect calls. To this end, we leverage MLTA [54] which is a state-of-the-art static analysis tool specifically designed to accurately resolve kernel indirect calls. In addition, we need a different feedback mechanism than coverage because the same `request_module()` invocation may actually correspond to different modules being loaded, *e.g.,* when the argument to the function is different. Therefore, on top of coverage, we additionally add the feedback of the actual module name that is being loaded by instrumenting `request_module()`. If a test case manages to gather new feedback in either metric, we will put the test in the corpus.

Table III: Breakdown of module loading fuzzing results

Distro	crypto	fs	net	tty	other	Sum	Total
Ubuntu	85	58	157	13	3	316	371
Fedora	69	47	106	11	3	236	272
Debian	97	52	132	11	7	299	361
Suse	99	52	145	15	0	311	465

Following the above setup, we performed an experiment by fuzzing the latest version of the three distros, *i.e.,* Ubuntu-22.04, Fedora-36, Debian-11.4, and SLE15-SP3 separately for three weeks (each with 16 CPU cores). In the end, we obtained 316 unprivileged loadable modules from Ubuntu kernel versus 236 in Fedora, 299 in Debian, and 311 in Suse. The breakdown is shown in Table III. The *Total* column in Table III represents the total number of unique modules that can be loaded from all `request_module()` sites in the core kernel. In particular, our fuzzer discovered 316 out of 371 (85.1%) unprivileged loadable modules with concrete test cases to trigger the module loading in Ubuntu kernel. As for Fedora, we discovered 236 out of 272 (86.7%) versus 299 out of 361 (82.8%) in Debian. Suse has the lowest ratio of unprivileged loadable modules – 311 out of 465 (66.8%), as it has the highest number of loadable modules. After collecting the unprivileged module loading results, SyzBridge builds a database of PoCs that can load the corresponding modules to support the privilege adaptation.

E. Privilege Adaptation for Kernel Security Checks

Modern Linux kernels have a feature called namespace [25] where an unprivileged user can create a namespace dynamically and make itself a privileged user within the namespace. This feature is commonly enabled in downstream kernels, *i.e.,* when `kernel.unprivileged_userns_clone` is set to `1`. When used, an unprivileged user can have the illusion that it is root user and can be granted any capabilities.

Therefore, if a PoC fails to pass certain security checks can now pass through them by running inside a namespace. In fact, prior exploits have leveraged such techniques to achieve privilege escalation (*e.g.,* [3] [2]). For example, a capability check is usually accomplished by `ns_capable()` and its variant functions, *e.g.,* `ns_capable(net->user_ns, CAP_NET_ADMIN)` verifies if the `net` namespace of the process contains `CAP_NET_ADMIN`. However, some kernel functionalities would verify the privilege of the calling process by its original identity rather than the one in the namespace, *e.g.,* `ns_capable(&init_user_ns, cap)`. This is similar for uid and gid checks.

We instrument the upstream kernel so that the exact check can be pinpointed along the execution. For each check function, we inspect the examined namespace by the check. If it is `init_user_ns`, we conclude that it is not possible for an unprivileged user to satisfy this check. In other cases, we perform the namespace adaptation with the highest privilege available, such as setting the uid and gid to 0 and granting all capabilities.

V. Evaluation

A. Experiment Design

We designed two experiments to evaluate: (1) the effectiveness of adapting upstream PoCs to downstream kernels using SyzBridge, using a general dataset of all KASAN bugs in the upstream, and (2) the effectiveness of bridging the gap of a state-of-the-art upstream exploitability assessment tool, SyzScope [70], using a specific dataset of high-risk upstream bugs.

For each syzbot bug, we select the candidate major release kernels based on the same two criteria as in the exploratory experiment: (1) the bug has not been patched in the downstream kernel, and (2) the corresponding major release is still under support when the bug is discovered. (Table VII shows the complete distro dataset we used for the evaluation).

Experiment I: Evaluate upstream PoC adaptations. We use the same dataset as in the exploratory experiment in §III. In total, we prepared 230 upstream KASAN bugs from syzbot, and 43 downstream distro releases from four major vendors: Ubuntu, Fedora, Debian, and Suse. We conducted all the experiments on the same machine as the exploratory experiment. We give 600 seconds for each attempt to run the PoC. Since we have run the PoC on the downstream kernel as root user and an unprivileged user before, we now attempt to run SyzBridge to improve the triggerability results. Specifically, if the PoC does not even trigger the bug as root, SyzBridge applies the environment-related adaptations accordingly to see if it can trigger as a result. If it is successful, SyzBridge then apply privilege-related adaptations to attempt to downgrade its privilege requirements. We record the results of the adaptations accordingly.

Experiment II: Evaluate exploitability assessment pipeline. To understand how many upstream high-risk bugs indeed affect downstream kernels, we conducted an additional experiment by feeding the results of SyzScope which turned 183 seemingly low-risk bugs into high-risk ones. Additionally, we use all KASAN write bugs from 2017 to 2022, which are considered as high-risk already. In total, the dataset constitutes a total number of 282 high-risk upstream bugs and 68 distros. We use the same experiment setup – 600 seconds for each attempt, 5 hours maximum for SyzScope (1 hour of fuzzing and 4 hours of symbolic execution).

B. Results of Experiment I

Table IV shows the breakdown of the results, similar to how we presented exploratory experiment results, we merged distro major releases into a single category by the vendor. We knew from the exploratory experiment that only 55, 58, 33 and 29 out of 230 bugs are triggered by Ubuntu, Fedora, Debian, and Suse respectively. In addition, only 2, 3, 2, and 1 bugs (0.4% - 1.3%) are triggered directly by an unprivileged user.

Surprisingly, looking at the final result after applying SyzBridge, we can see from the column “After Adaptation” that 86, 77, 63, and 57 bugs can be triggered by root, representing an average of 27 additional bugs triggered per distro vendor, an improvement of 61%. More importantly, 27, 46, 21, and 18 bugs can be triggered by an unprivileged user, representing an average of 26 additional bugs whose privilege requirement is downgraded, an improvement of 1,300%.

To attribute the success to the various types of adaptations, we count how many times a particular type of adaptation helped in Table IV. Regarding the environment adaptation, we find that the distro background noise related adaptations are the most common (EA2) and the adaptations for failed preparation steps (EA1) are the least common. The high EA2 count illustrates how different the environment of a distro OS is compared to a minimal OS configured specifically for fuzzing (on syzbot). In particular, Ubuntu has a significant number of bugs that fail to reproduce because of the distro noise. We investigated the cause and it turns out that the major reason is that Ubuntu comes with a package pre-installed called snap [15] while the other three vendors did not. It is a software packaging and development system that helps rapidly deploy software without worrying about dependencies, it is an official piece of software developed by Canonical [1]. To install snap packed app, it has to be first decompressed and then mounted through loop devices [11]. Due to the limited number of loop devices, Ubuntu pre-installs a number of snap apps that occupy most of the existing loop devices right after boot and leave other processes (*e.g.,* PoC) unable to access these critical resources. Because snap is only installed and used on Ubuntu distros, we can see that the other two distros suffer less from the resource competition issue (there are still other system processes that occupy a subset of loop devices).

Kernel module missing (EA3) is the second most common type of environment adaptation that is helpful, we see 9, 4, 21, and 11 bugs that have become triggerable after SyzBridge load the required modules. It is interesting to note that the number of bugs that require additional modules on Ubuntu is twice as many as those on Fedora. To make sense of the discrepancy, we find that the total numbers of loadable kernel modules, including those that cannot be loaded through request_module(), on these two distros are ∼6,000 for Ubuntu-22.04 versus ∼4,000 for Fedora-36 respectively, indicating that Ubuntu has a larger attack surface and explains why more bugs can be adapted through module loading. Note that the number of modules loadable by an unprivileged user is substantially small (estimated in Table III). Debian has the highest number of EA3 adaptations. Upon investigation, we find that it is due to Debian's kernel configuration setting for CONFIG_BLK_DEV_LOOP, which is set to m. This configuration indicates that the loop device will be compiled into a module. As a result, Debian needs to load the loop module when required. This specific reason accounts for 17 out of the 21 EA3 adaptations observed in Debian. Similarly, Suse also experiences the same cause, with 6 out of the 11 EA3 adaptations due to loading loop modules.

Note that a single bug may affect multiple release versions of a distro, such as Ubuntu-20.04 and Ubuntu-18.04. It is possible that certain releases require adaptations while others can directly reproduce the bug. In such cases, the bug will be counted in both the ”Before Adaptation” column and the ”Environment Adaptation” column. Therefore, simply adding the numbers in ”Before Adaptation - root” with the ”Environ-

Table IV: Experiment I overall results

Distro	Before Adaptation		Environment Adaptation				Privilege Adaptation			After Adaptation	
	root	normal	Total Bugs	EA1	EA2	EA3	Total Bugs	PA1	PA2	root	normal
Ubuntu	55	2	46	3	36	9	26	5	26	86	27
Fedora	58	3	20	3	14	4	43	1	42	77	46
Debian	33	2	30	4	24	21	18	2	18	63	21
Suse	29	1	24	0	19	11	17	0	17	57	18

EA1: Environment adaptation for failed preparation steps. EA2: Environment adaptation for background noise.
EA3: Environment adaptation for module loading.
PA1: Privilege adaptation for module loading. PA2: Privilege adaptation for kernel security checks.

ment Adaptation - Total Bugs” may result in a higher number than the ”After Adaptation - root”. This rule also applies to ”Privilege Adaptation”.

Moving on to privilege adaptation, adaptations for kernel security checks (PA2) are the most common. This is due to more and more features becoming “unavailable” to a regular user, which is supposed to reduce the attack surface. Nevertheless, it turns out that many of such “protected” features can become readily available as long as the namespace is enabled (which is the case in most modern Linux distros). Adaptations for unprivileged module loading (PA1) are less common but are still helpful. Recall that there are bugs that require EA3 adaptations to load additional modules. With the help of the database we built from the results of guided fuzzing (mentioned in §IV-D), we found a subset of those cases where we can load such modules as an unprivileged user. This step is necessary in two of the six real-world CVEs as well as other high-risk but unexploited bugs, which we will discuss in §V-C.

Some bugs, despite not having privilege requirements, may fail to reproduce due to environmental issues. In these cases, only environment adaptation is needed to be reproduced by a normal user. For example, in Table IV, Debian initially had 2 bugs reproducible by a normal user, and after privilege adaptation of 18 bugs, it increased to 21 bugs, with one bug requiring only environment adaptation.

False Positives. SyzBridge runs a concrete test case to reproduce a kernel bug, and thus it in theory should not have any false positives by design. SyzBridge does instrument the downstream kernels so that it can capture KASAN bugs. This requires recompiling the kernel and enabling a few kernel config options such as `CONFIG_KASAN`, `CONFIG_DEBUG_KERNEL`, *etc.*. We are aware that occasionally sanitizers will report false positives due to sanitizer bugs [7]. Nevertheless, we consider such issues to be out of the responsibility of SyzBridge. Instead, we assume sanitizers to be correctly implemented in this project.

False Negatives. We sample 20 bugs that fail to trigger on some downstream kernels. For each bug, we sample between one to three downstream kernels to investigate. In total, this gives us 40 bug-distro pairs. After our manual investigation by cross-checking the vulnerable commit and code snippets, we have identified 36 out of 40 pairs are true negatives, which indicates the bug does not exist in the target distro. The four false negatives fall under two reasons which we summarize below.

Table V: Experiment II overall results

Distro	Before Adaptation		After Adaptation	
	root	normal	root	normal
Ubuntu	54	1	82	35
Fedora	48	3	82	50
Debian	47	3	84	31
Suse	27	1	42	9

(1) Failing to win the race within a given time limit. As we mentioned in §IV, downstream OSes contain many more user space processes and services than upstream. The worse background noise results in difficulty in winning a race, so the PoC requires a longer repeat execution time. However, we set a time limit of 600 seconds for each bug-triggering attempt to make SyzBridge scalable. If the PoC still cannot win the race within 600 seconds, we will give up this case and consider it failed. To know how many bugs failed due to this problem, we ran a separate experiment for these bugs that extended the time limit to 24 hours and noticed 1 PoC can successfully trigger the bug in an hour and another PoC triggered the bug after 23 hours. Note that we only limit the reproducing time because of the sheer number of bug/distro pairs in our experiment. However, we envision the limit can be lifted easily in a real deployment setting where we run SyzBridge with syzbot side by side. That way, only a few bugs need to be analyzed per day on average.

(2) Failing to extract PoC execution trace. We find two pairs where we did not successfully obtain the execution trace from the upstream kernel. This results in a failure of our missing module analysis, and further impacts the adaptation. The reason is due to kernel crashes sometimes, resulting in partial losses of the execution trace. This is something we can potentially overcome by improving the way `ftrace` collects the execution traces.

C. Results of Experiment II

Out of the 282 high-risk upstream bugs, we found that, on average, 44 (15.6%) bugs could be reproduced on at least one distribution, without any adaptation provided by SyzBridge. However, only an average of 2 (0.7%) bugs could be reproduced with normal privileges, which is a crucial requirement for an exploitable bug. Table V shows the detailed results before and after applying SyzBridge. Without

Table VI: High-risk bugs from experiment II

Bug	Bug Primitive	Affect	Before Adaptation		Environment Adaptation			Privilege Adaptation		After Adaptation	
			root	normal	EA1	EA2	EA3	PA1	PA2	root	normal
CVE-2022-27666*	OOB W	UFD	F	-	-	UD	UD	UD	UFD	-	UFD
CVE-2022-0185	OOB W	UFD	UFD	-	-	-	-	-	UFD	-	UFD
CVE-2021-22600	DF	UFD	UFD	-	-	-	-	-	UFD	-	UFD
CVE-2021-22555	OOB W	UFD	F	-	-	UD	UD	UD	UFD	-	UFD
CVE-2021-4154	CFH	UFD	UFD	-	-	-	-	-	UFD	-	UFD
CVE-2021-3715	UAF W	U	U	-	-	-	-	-	U	-	U
cf7393b*	UAF W	UFD	UFD	-	-	-	-	-	UF	D	UF
4b0830a	UAF W	UFD	U	D	-	-	F	-	F	U	FD
e67f2fc	UAF W	UFD	-	-	-	UFD	-	-	UFD	-	UFD
2389bfc*	CFH	UFD	UFD	-	-	-	-	-	F	UD	F
403eb21	CFH	UFS	UF	-	S	-	-	-	UF	S	UF
f4c90f2*	OOB W	UFDS	UFDS	-	-	-	-	-	UFS	D	UFS
380acd1*	DF	UF	UF	-	-	-	-	-	UF	-	UF
b53aed2	OOB W	UFD	UFD	-	-	-	-	-	UFD	-	UFD
e2d0f38	CFH	F	-	-	-	F	-	-	F	-	F
d35e6e8	NPD W	UFD	UFD	-	-	-	-	-	UFD	-	UFD
60e3243	CVW	UFDS	UFDS	-	-	-	-	-	UFDS	-	UFDS
a53b68e	OOB W	UF	F	-	-	U	U	U	UF	-	UF
5ad0e07	NPD W	UF	UF	-	-	-	-	-	UF	-	UF
7c7245f	OOB W	UFD	UFD	-	-	-	-	-	UFD	-	UFD
f1834e1	CFH	FD	-	-	D	FD	-	-	-	-	FD
ed87cd6	CFH	F	F	-	-	-	-	-	F	-	F
e4c5c37	AVW	FD	-	-	-	FD	-	-	F	D	F
e3e31b1	UAF W	FD	F	-	-	D	-	-	F	D	F
b8febdb	UAF W	FD	-	-	-	-	FD	FD	FD	-	FD
ba1aecb	DF	FD	F	-	-	D	-	-	FD	-	FD
955089c	CFH	FD	FD	-	-	-	-	-	FD	-	FD
457491c	CFH	UFD	-	-	UFD	-	UFD	UFD	UFD	-	UFD
6578348	CFH	UFDS	UD	FS	-	-	-	-	UD	-	UFDS
26de18d	CVW	UFD	-	-	U	UFD	-	-	UF	D	UF
418578d	UAF W	UFD	-	-	UD	UFD	-	-	F	-	UFD
2a62245	OOB W	F	F	-	-	-	-	-	F	-	F
232223b	UAF W	FS	FS	-	-	F	-	-	F	S	F
27934d2	UAF W	UFDS	UFDS	-	-	-	-	-	UFS	D	UFS
26cb120	DF	UFD	UFD	-	-	-	-	-	UF	D	UF
0c4fd9c	AVW	UFDS	UFDS	-	-	-	-	-	UFDS	-	UFDS
f99edae	UAF W	UFD	UFD	-	-	-	-	-	U	FD	U
d020174	UAF W	UD	-	-	U	-	-	-	U	D	U
db84232	DF	D	-	-	D	-	-	-	-	-	D
5bb09c0	AVW	UFDS	UFS	-	D	-	-	-	UFDS	-	UFDS
7be8b46	CVW	D	-	-	-	D	-	-	-	-	D
c897760	UAF W	D	-	-	-	D	-	-	-	-	D
f381dee	AAW	UFDS	UFS	D	-	-	-	-	-	UFS	D
481c3fd	UAF W	UFDS	UFDS	-	-	-	-	-	FS	U	FDS
8393c02	OOB W	UDF	UDF	-	-	-	-	-	UF	D	UF
551ff46	UAF W	UF	F	-	-	-	U	-	F	U	F
80f661b	OOB W	F	-	F	-	-	-	-	-	-	F
264bca3	OOB W	UFD	-	UFD	-	-	-	-	-	-	UFD
d425214	DF	UFD	FD	-	-	U	-	-	-	FD	U
522643a	UAF W	UFDS	UFDS	-	-	-	-	-	F	UDS	F
a0d209a	DF	DF	DF	-	-	-	-	-	DF	-	DF
34b3d29	UAF W	F	F	-	-	-	-	-	F	-	F

* indicates the bugs we developed exploits for
U: Ubuntu, F: Fedora, D: Debian, S: Suse
OOB W: out-of-bounds write, UAF W: use-after-free write, DF: double free, CFH: control flow hijacking
NPD W: null-ptr-defer write, CVW: constrained value write (of a specific variable)
AVW: arbitrary value write, AAW: arbitrary address write
EA1: Environment adaptation for failed preparation steps. EA2: Environment adaptation for background noise.
EA3: Environment adaptation for module loading
PA1: Privilege adaptation for module loading. PA2: Privilege adaptation for kernel security checks

SyzBridge, SyzScope's results can confirm only 54, 48, 47, and 27 root-triggerable bugs for Ubuntu, Fedora, Debian, and Suse, respectively. Furthermore, only 1, 3, 3, and 1 bugs are triggerable by normal privileges — this result delivers a misleading impression of the exploitability of such bugs in the downstream Linux distros. However, the evaluation results significantly improved after we applied SyzBridge. Specifically, SyzBridge adapted an additional 28, 34, 37, and 15 PoCs for Ubuntu, Fedora, Debian, and Suse, respectively, successfully triggering the corresponding bugs. This enhancement resulted in an improvement of 51%, 70%, 78%, and 56% for each distribution. Impressively, SyzBridge also downgraded the privilege requirements for 29 bugs on average per distro, improving the results by 15 times — more than an order of magnitude. It changes the total number of high-risk "downstream-applicable" bugs from 4 to 53 across all four distros. Clearly, by integrating with SyzBridge, SyzScope's results become much more relevant to downstream kernels in the real world.

We list all 53 bugs and their affected downstream distro vendors in Table VI. They include all of the 6 existing CVEs we gathered from the Internet. They originate from syzbot upstream bugs, and have been exploited on downstream distros; in other words, none were missed by SyzBridge. Note that one of the CVEs is credited to our efforts, where we successfully developed an end-to-end exploit against the latest Ubuntu distro . Furthermore, the remaining 47 high-risk "downstream-applicable" bugs have not been assigned CVEs to this day, which are potentially weaponizable (we will discuss our efforts in attempting to turn them into exploits later in this section). The results indicate that indeed many potentially exploitable bugs (relevant to downstream distros) were neglected by exploitability assessment tools and human experts. Therefore, it demonstrates the value of SyzBridge and its important role in exploitability assessment in the Linux kernel ecosystem.

In particular, 9 bugs have a control flow hijack primitive, which is a very strong primitive that can be exploited through return-oriented programming for instance. In addition, there are 16 use-after-free (UAF) write bugs, 11 out-of-bound (OOB) write bugs, and 7 double-free bugs, which are all highly likely exploitable as well.

If we look at Table VI more closely, we can see that only 5 bugs are triggered directly without SyzBridge by an unprivileged user. All other cases went through various adaptations to succeed. In particular, 10 pairs required module loading to become triggerable by an unprivileged user, and 91 pairs require namespace. Note that when both adaptations are listed in the table, it means that both are required for the bug to trigger.

End-to-end exploitation. To validate the exploitability of the bugs identified through our integrated pipeline, we sampled 5 bugs from Table VI that were previously not known to be exploitable publicly — they are annotated with the ∗ symbol. The five cases include the one CVE case that is assigned due to our reporting. We succeeded in exploiting all five of them and achieving privilege escalation in one of the affected downstream distros. This sample consisted of one open (unfixed) bug at the time and four other fixed bugs. The open bug has a write primitive and SyzBridge adapted it successfully to reproduce on the Ubuntu distro. We successfully developed an end-to-end exploit that bypasses all the kernel mitigation and escalates the local privilege on the latest Ubuntu kernel. We later reported it to the Ubuntu maintainers and CVE association, the detail of how we handle this vulnerability is in $VI-D. As for the four fixed bugs, we chose to develop simplified end-to-end exploits by disabling kernel defenses such as *KASLR*, *smap*, and *smep*, as proof-of-exploitability.

```
 1 net/sched/cls_api.c:
 2 static int tc_new_tfilter(struct sk_buff *skb,
 3   struct nlmsghdr *n, struct netlink_ext_ack *extack)
 4 {
 5         struct net *net = sock_net(skb->sk);
 6         if (!netlink_ns_capable(skb, net->user_ns,
                CAP_NET_ADMIN))
 7                 return -EPERM;
 8         err = __tcf_qdisc_find(net, &q, &parent, t->
                tcm_ifindex, false, extack);
 9         if (err)
10                 return err;
11         vulnerable_func();
12 }
13
14 net/ipv4/ipip.c:
15 static int __net_init ipip_init_net(struct net *net)
16 {
17         dev = alloc_netdev(ops->priv_size, name,
                NET_NAME_UNKNOWN, ops->setup);
18         if (!dev) {
19                 err = -ENOMEM;
20                 goto failed;
21         }
22         err = register_netdevice(dev);
23 }
24
25 module_init(ipip_init);
26 ...
```

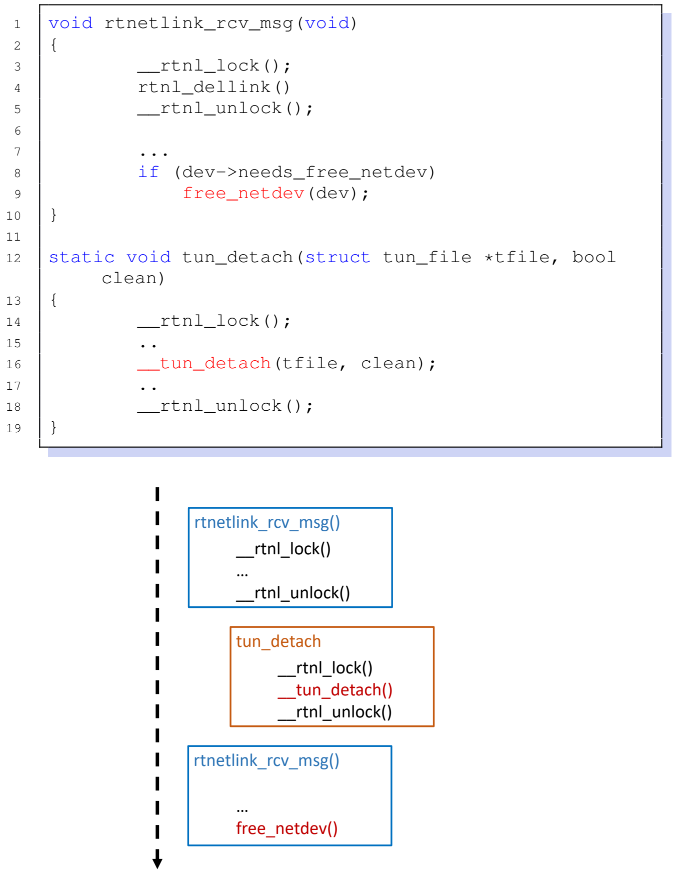
Figure 3: Case study of bd699d3

In summary, we demonstrate the power of the automation and effectiveness of SyzBridge in analyzing the bug exploitability against downstream kernels.

D. Case Study

bd699d3 Figure 3 shows two functions: `tc_new_tfilter()` and `ipip_init_net()`. The first function is located in the core part of the kernel (as opposed to a kernel module), and the second is located in a kernel module called `ipip`. The bug is related to the first function. To trigger the bug, it is necessary to reach line 11 (labeled as `vulnerable_func()`). Before line 11, we can see there are two checks at line 6 and line 9 that need to be cleared.

Initially, when we execute the PoC on a downstream kernel, Ubuntu-20.04.2 in this case, as root user, we find that it fails the second check (line 9), aborting the execution early. Looking at it more closely, we realize that `__tcf_qdisc_find()` attempts to retrieve a network device from a global `net_device` linked list by a specific id. Normally, a network device will be put into this linked list during the `module_init` function when initializing the module. In syzbot, it compiles upstream kernels by statically linking all kernel modules (most commonly used ones) into



```
1   void rtnetlink_rcv_msg(void)
2   {
3           __rtnl_lock();
4           rtnl_dellink()
5           __rtnl_unlock();
6
7           ...
8           if (dev->needs_free_netdev)
9               free_netdev(dev);
10  }
11
12  static void tun_detach(struct tun_file *tfile, bool
        clean)
13  {
14          __rtnl_lock();
15          ..
16          __tun_detach(tfile, clean);
17          ..
18          __rtnl_unlock();
19  }
```

Figure 4: Case study of e67f2fc

a monolithic kernel binary [20], instead of compiling them into individual loadable modules (*i.e.,* separate .ko files). As a result, every network device is automatically put into this global net_device linked list during the kernel boot process. In contrast, Ubuntu-20.04.2 does not statically link most network device kernel modules into the kernel binary and instead maintain them as separate loadable modules. To trigger the bug, we simply need to load the requested network device ipip before running the PoC, *e.g.,* through modprobe, the module will be initialized. As shown in Figure 3, function ipip_init_net() that will register a network device and register it through the global net_device linked list. Because this buggy code is not tied to the module functionality, as long as it passes the check at line 9, the bug will be triggered.

e67f2fc. As shown in Table VI, this bug requires environment adaptation for background noises in order to trigger the bug on downstream distros. More specifically, Figure 4 shows that when the vulnerable object is destroyed by __tun_detach at line 16, another kernel thread might still be using it by free_netdev() at line 9. As shown in the interleaving graph (bottom half of Figure 4), in order to trigger the bug, the thread executing tun_detach() must be scheduled to go right after the lock has been released at line 5 and before free_netdev() is executed at line 9. Surprisingly, the upstream kernel seems to win the race easily, with no loop to repeat the race. However, the same PoC cannot win the race in downstream distros. When SyzBridge executes it one-time and observes that it did not trigger the bug, it forces the PoC to repeat until either the bug is triggered or a timeout is reached. On the downstream kernels, such a simple adaptation causes the race condition to be triggered in less than a second.

VI. Limitations and Discussion

A. Missing kernel traces

Aside from the engineering issue of not being able to collect the complete execution traces sometimes, there is another important decision that may cause the traces to be incomplete in our current design. Specifically, SyzBridge currently inspects only the traces corresponding to the PoC process and ignores all other traces. This is to avoid unrelated modules being considered relevant to the bug. However, kernel bugs are not always triggered within the PoC process. Sometimes they can also be triggered by kernel worker threads. Worker threads are kernel internal threads, they can be created on demand or long-running. In either case, it would not have the same pid as that of the PoC process. If a bug is triggered in such kernel threads, SyzBridge cannot tell which kernel worker thread is the one related to the bug and will not load the corresponding kernel module if missing in downstream kernels. In our dataset, we have not encountered this issue. However, to support this, one potential solution is to learn from the upstream bug report which does contain some information (*e.g.,* call stack) about the kernel thread that triggered the bug. This can be used to locate the relevant kernel thread to be included in our analysis.

B. Unprivileged module loading

Because the search process provided by fuzzing is opportunistic in nature, it is inevitable to have false negatives. In other words, we might miss modules that can actually be loaded with an unprivileged user. Currently, we constrain the search space of syscalls that can potentially reach request_module(). However, since we rely on existing syzkaller descriptions which encode a limited set of syscalls and arguments, we are inherently limited by the coverage of these descriptions. Indeed, many request_module() are never reached during our fuzzing session. As future work, we can incorporate more input sources such as the Linux testing project that can potentially improve the odds of reaching request_module(). This has been attempted by several prior work [56], [50], [33], [35].

C. Code context change

We initially anticipated many cases of code context changes that impact the triggerability of the bug, requiring adjustments of the test case, *e.g.,* changing a constant of a syscall argument. Indeed, a recent project [45] proposed to make adjustments to an exploit so that it can succeed on a different kernel version. However, the project simply assumes the bug itself is always triggered by the exploit and focuses on the adaptation of the post-bug-triggering logic of the exploit. Exploits require additional steps such as heap feng shui, converting one primitive to another, and bypassing defenses such as KASLR. They are much more complicated than bug-triggering PoCs, and therefore code context changes affect the exploits much more. When we look at the results of this project, indeed it is rare that PoCs fail due to code context changes.

D. Ethics consideration

For the 1-day vulnerability that we successfully exploited, we did not publicly release any information regarding the exploit until after we responsibly disclosed it to a CVE authority (which led to a CVE number assigned promptly) and all affected distro vendors patched the vulnerability.

E. Integration with other bug assessment systems

While SyzBridge can be helpful to reveal the triggerability and privilege requirement of upstream bugs on downstream distros, it does not tell the full story of bug impacts. We envision SyzBridge integrated with other automated bug impact analysis systems for fuzzer-exposed bugs [70], [32], [66]. In fact, we already made a small integration with SyzScope [70] that revealed more exploitable primitives of given bugs. Furthermore, SyzBridge can also work with existing automatic exploit generation systems like KOOBE [32] and FUZE [66]. Such an automated pipeline can provide a comprehensive exploitability assessment capability that assists security researchers in understanding the capabilities of a bug.

F. Upstream patch propagation

It is important to note that affected downstreams must promptly port the patches, even if the upstream has already patched the bugs identified by syzbot. This is because the downstreams are the ones actually utilized by the end users. Unfortunately, we know that the propagation of upstream patches to downstreams can be time-consuming. It has been shown that the patch delay between upstream and downstream kernels (i.e., Android) is often months and even years. This is because patch porting is a labor-intensive and error-prone process where downstream maintainers need to understand the patch and its impact, and even adapt it to their own code base. They also need to ensure that the whole system, after applying the patch, still works as expected. In the case of the 53 bugs shown in Table VI, we find that 38 of them do not need any adaptation and can therefore be technically ported quickly. There have been some recent progress on automatically determining the correctness and safety of patches [38], [49], [55], as well as adapting a patch when it is ported [60], [59]. Such techniques can potentially be helpful shortening the patch propagation delay.

VII. Related Work

Cross-version Bug and Exploit Assessment. A large body of work focused on detecting recurring bugs [67], [44], [57], [47]. They aim to statically determine, based on various code similarity metrics, whether a given bug in a reference code base exists in a target code base. However, such approaches are imprecise in nature and cannot provide a concrete PoC to actually trigger the bug. VulScope [36] offers a dynamic approach to migrate the PoC to a different version of the user-space software that triggers the same bug. It utilizes the crashing trace from the reference version of the software to guide the fuzzing mutation. But their solution aims at user-space programs instead of the Linux kernel. In fact, as we show in this paper, the failure reasons and required adaptations identified in the kernel are completely different from what is needed for user-space programs. AEM [45] does aim at the Linux kernel. However, its goal is to migrate exploits as opposed to bug PoCs. This is an important distinction because exploits are much longer than PoCs and thus more likely to require adaptations. In fact, AEM assumes that the original exploit can successfully trigger the bug on the target kernel without adaptations, and only tries to adapt the post-bug-triggering logic. Therefore, it is orthogonal to what SyzBridge addresses.

Bug Exploitability Assessment. Some prior works [58], [61], [62] make use of qualitative metrics to score bug primitive. Evocatio [46] leverage a capability-guided fuzzer to uncover new bug primitives for bugs in user-space software; The results show that 19 out of 38 bugs have more serious primitives, resulting in their CVSS scores being raised. SyzScope [70] aims to uncover primitives for Linux kernel bugs, by combining kernel fuzzing, static analysis, and symbolic execution. In the end, SyzScope escalate 183 kernel low-risk bugs to high-risk bugs and discovered a total of 4,800 new high-risk primitives for the 183 bugs.

Automated Exploit Generation. Brumley's work [30] first introduced automated exploit generation based on patches (AEGP). Averions proposed the AEG concept [29] in 2014. Revery [65] leverages concolic execution and memory modeling to make AEG scalable. For user space program, AngErza [37] and Huang's work [43] use symbolic execution to module the bug primitives of real-world programs. In the context of the Linux kernel, FUZE [66] targets Use-after-free bugs specifically, while KOOBE [32] focuses on heap out-of-bound write bugs. They both utilize fuzzing and symbolic execution to explore new paths and discover exploitable states such as arbitrary memory write, and control flow hijacking. K-LEAK [51] attempts to automate the generation of infoleak exploits given fuzzer-exposed bugs with PoCs, which are often necessary to achieve end-to-end privilege escalation.

VIII. Conclusion

This paper is motivated by our attempt to understand why an upstream PoC cannot trigger bugs in downstream Linux distributions. Through our small-scale exploratory experiment, we develop an automated system named SyzBridge that can adapt the PoC to satisfy the requirements, resulting in not only successful bug triggering but also downgrading of privilege requirement. Our evaluation of 230 upstream bugs shows that our adaptation is effective as it successfully turned 61% more bugs triggerable by root and 13 times more bugs triggerable by an unprivileged user.

Acknowledgment

We thank anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Science Foundation under Grant #1652954, #1953933 and #2155213.

References

[1] Canonical. https://canonical.com/.

[2] CVE-2022-0185. https://www.willsroot.io/2022/01/cve-2022-0185.html.

[3] CVE-2022-27666. https://etenal.me/archives/1825.

[4] Debian distribution. https://www.debian.org/.

[5] Debian kernel source. https://kernel-team.pages.debian.net/kernel-handbook/ch-source.html.

[6] Fedora distribution. https://getfedora.org/.

[7] Finding bugs with sanitizers. https://lwn.net/Articles/909245/.

[8] Ftrace. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[9] Kasan: use-after-free write in null_skcipher_crypt. https://syzkaller.appspot.com/bug?id=517fa734b92b7db404c409b924cf5c997640e324.

[10] kctf. https://google.github.io/kctf/vrp.html.

[11] loop device. https://man7.org/linux/man-pages/man4/loop.4.html.

[12] The lwn.net linux distribution list. https://lwn.net/Distributions/.

[13] public kctf responses. https://docs.google.com/spreadsheets/d/e/2PACX-1vS1REdTA29OJftst8xN5B5x8iIUcxuK6bXdzF8G1UXCmRtoNsoQ9MbebdRdFnj6qZ0Yd7LwQfvYC2oF/pubhtml.

[14] request_module. https://www.kernel.org/doc/htmldocs/kernel-api/API---request-module.html.

[15] snap intro. https://ubuntu.com/core/services/guide/snaps-intro.

[16] Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v4.10/process/submitting-patches.html.

[17] Suse distribution. https://www.suse.com/.

[18] syzbot. https://github.com/google/syzkaller/blob/master/docs/syzbot.md.

[19] Syzbot - Linux 4.19. https://syzkaller.appspot.com/linux-4.19.

[20] Syzbot kernel config. https://syzkaller.appspot.com/text?tag=KernelConfig&x=a53fd47f16f22f8c.

[21] syzkaller/prog/minimization.go. https://github.com/google/syzkaller/blob/master/prog/minimization.go.

[22] Ubuntu distribution. https://ubuntu.com/.

[23] Ubuntu Kernel Delta. https://wiki.ubuntu.com/Kernel/FAQ/UbuntuDelta.

[24] Usb-rawgadget. https://docs.kernel.org/usb/raw-gadget.html.

[25] User namespace. https://lwn.net/Articles/420624/.

[26] zdi. https://www.zerodayinitiative.com/.

[27] Kernel addresssanitizer. https://github.com/google/kasan, 2020.

[28] Kmsan. https://github.com/google/kmsan, 2020.

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

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

[31] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, 2021.

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

[33] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1165–1184, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[35] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

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

[37] Shruti Dixit, T K Geethna, Swaminathan Jayaraman, and Vipin Pavithran. Angerza: Automated exploit generation. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6, 2021.

[38] Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, 2022.

[39] Google. Kcsan. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html, 2020.

[40] Google. syzbot. https://syzkaller.appspot.com/upstream/, 2020.

[41] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *IEEE Symposium on Security and Privacy*, 2023.

[42] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. Demystifying the dependency challenge in kernel fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 2022.

[43] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[44] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

[45] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. Aem: Facilitating cross-version exploitability assessment of linux kernel vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[46] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring bug capabilities from a single poc. 2022.

[47] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[48] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *Network and Distributed System Security Symposium*, 2013.

[49] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 524–535, 2019.

[50] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In *31st Annual Network and Distributed System Security Symposium, NDSS*, 2024.

[52] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095, 2022.

[53] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[54] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th*

ACM Conference on Computer and Communications Security (CCS), London, UK, 2019.

[55] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1562–1579, 2020.

[56] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: optimizing os fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium*, 2018.

[57] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, 2015.

[58] Mustafizur R. Shahid and Hervé Debar. Cvss-bert: Explainable natural language processing to determine the severity of a computer security vulnerability from its description. *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1600–1607, 2021.

[59] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. Automated patch backporting in linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, 2021.

[60] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. Automated patch transplantation. *ACM Trans. Softw. Eng. Methodol.*, 2021.

[61] Georgios Spanos and Lefteris Angelis. A multi-target approach to estimate software vulnerability characteristics and severity scores. *Journal of Systems and Software*, 146:152–166, 2018.

[62] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. Assessment of vulnerability severity using text mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, PCI 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[63] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[64] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[65] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1914–1927, New York, NY, USA, 2018. Association for Computing Machinery.

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

[67] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[68] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, Boston, MA, August 2022. USENIX Association.

[69] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, Boston, MA, August 2022. USENIX Association.

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

APPENDIX

	Distro Major Release	Code Name	Kernel Version	Released Date	End of Support date
Ubuntu	Ubuntu-16.04.2	xenial	4.4.62	Feb 16 2017	Apr 27 2021
	Ubuntu-16.04.3	xenial	4.4.87	Aug 3 2017	Apr 27 2021
	Ubuntu-16.04.4	xenial	4.4.116	Mar 1 2018	Apr 27 2021
	Ubuntu-16.04.5	xenial	4.4.131	Aug 2 2018	Apr 27 2021
	Ubuntu-16.04.6	xenial	4.4.142	Feb 28 2019	Apr 27 2021
	Ubuntu-16.04.7	xenial	4.4.186	Aug 13 2020	Apr 27 2021
	Ubuntu-18.04	bionic	4.15.20	Apr 26 2018	-
	Ubuntu-18.04.1	bionic	4.15.29	Jul 26 2018	-
	Ubuntu-18.04.2	bionic	4.15.45	Feb 15 2019	-
	Ubuntu-18.04.3	bionic	4.15.55	Aug 8 2019	-
	Ubuntu-18.04.4	bionic	4.15.76	Feb 12 2020	-
	Ubuntu-18.04.5	bionic	4.15.112	Aug 13 2020	-
	Ubuntu-18.04.6	bionic	4.15.157	Sep 17 2021	-
	Ubuntu-20.04	focal	5.4.26	Apr 23 2020	-
	Ubuntu-20.04.1	focal	5.4.42	Aug 6 2020	-
	Ubuntu-20.04.2	focal	5.4.65	Feb 4 2021	-
	Ubuntu-20.04.3	focal	5.4.81	Aug 26 2021	-
	Ubuntu-20.04.4	focal	5.4.100	Feb 24 2022	-
	Ubuntu-20.04.5	focal	5.4.125	Sep 1 2022	-
	Ubuntu-22.04	jammy	5.15.25	Apr 21 2022	-
	Ubuntu-22.04.1	jammy	5.15.43	Aug 11 2022	-
Fedora	Fedora-26	26	4.11.8	Jul 11 2017	May 29 2018
	Fedora-27	27	4.13.9	Nov 14 2017	Nov 30 2018
	Fedora-28	28	4.16.3	May 1 2018	May 28 2019
	Fedora-29	29	4.18.16	Oct 30 2018	Nov 26 2019
	Fedora-30	30	5.0.9	May 7 2019	May 26 2020
	Fedora-31	31	5.3.7	Oct 29 2019	Nov 24 2020
	Fedora-32	32	5.6.6	Apr 28 2020	May 25 2021
	Fedora-33	33	5.8.16	Oct 27 2020	Nov 30 2021
	Fedora-34	34	5.11.12	Apr 27 2021	Jun 7 2022
	Fedora-35	35	5.14.10	Nov 2 2021	-
	Fedora-36	36	5.17.5	May 10 2022	-
Debian	Debian-9.0	stretch	4.9.30	Jun 17 2017	July 18 2020
	Debian-9.1	stretch	4.9.30	Jul 22 2017	July 18 2020
	Debian-9.2	stretch	4.9.51	Oct 7 2017	July 18 2020
	Debian-9.3	stretch	4.9.65	Dec 9 2017	July 18 2020
	Debian-9.4	stretch	4.9.82	Mar 10 2018	July 18 2020
	Debian-9.5	stretch	4.9.110	Jul 14 2018	July 18 2020
	Debian-9.6	stretch	4.9.130	Nov 10 2018	July 18 2020
	Debian-9.7	stretch	4.9.130	Jan 23 2019	July 18 2020
	Debian-9.8	stretch	4.9.144	Feb 16 2019	July 18 2020
	Debian-9.9	stretch	4.9.168	Apr 27 2019	July 18 2020
	Debian-9.10	stretch	4.9.189	Sep 7 2019	July 18 2020
	Debian-9.11	stretch	4.9.189	Sep 8 2019	July 18 2020
	Debian-9.12	stretch	4.9.210	Feb 2 2020	July 18 2020
	Debian-10.0	buster	4.19.37	Jul 6 2019	Sep 10 2022
	Debian-10.1	buster	4.19.67	Sep 7 2019	Sep 10 2022
	Debian-10.2	buster	4.19.67	Nov 16 2019	Sep 10 2022
	Debian-10.3	buster	4.19.98	Feb 8 2020	Sep 10 2022
	Debian-10.4	buster	4.19.118	May 9 2020	Sep 10 2022
	Debian-10.5	buster	4.19.132	Aug 1 2020	Sep 10 2022
	Debian-10.6	buster	4.19.146	Sep 29 2020	Sep 10 2022
	Debian-10.7	buster	4.19.160	Dec 5 2020	Sep 10 2022
	Debian-10.8	buster	4.19.171	Feb 6 2021	Sep 10 2022
	Debian-10.9	buster	4.19.181	March 27 2021	Sep 10 2022
	Debian-10.10	buster	4.19.194	June 19 2021	Sep 10 2022
	Debian-10.11	buster	4.19.208	Oct 9 2021	Sep 10 2022
	Debian-10.12	buster	4.19.235	March 26 2022	Sep 10 2022
	Debian-11.0	bullseye	5.10.46	Aug 14 2021	-
	Debian-11.1	bullseye	5.10.70	Oct 9 2021	-
	Debian-11.2	bullseye	5.10.84	Dec 18 2021	-
	Debian-11.3	bullseye	5.10.106	March 26 2022	-
	Debian-11.4	bullseye	5.10.127	Jul 9 2022	-
Suse	SLE-15-Initial	15	4.12.14-23	Jul 16 2018	Dec 31 2019
	SLE-15-SP1	15	4.12.14-195	Jun 24 2019	Jan 31 2021
	SLE-15-SP2	15	5.3.18-22	Jul 21 2020	Dec 31 2021
	SLE-15-SP3	15	5.3.18	Jun 22 2021	Dec 31 2022
	SLE-15-SP4	15	5.14.21	Jun 21 2022	-

- indicates that the distro is still remain valid up until the date we submit this paper

Table VII: Distro Dataset