When Top-down Meets Bottom-up: Detecting and Exploiting Use-After-Cleanup Bugs in Linux Kernel

Lin Ma*, Duoming Zhou*, Hanjie Wu†, Yajin Zhou*§, Rui Chang*, Hao Xiong*, Lei Wu*, Kui Ren*
*{linma, duoming, yajin_zhou, crix1021, mart1n, lei_wu, kuiren}@zju.edu.cn, Zhejiang University
†hanjiew@andrew.cmu.edu, Carnegie Mellon University

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179356

***Abstract*—When a device is detached from the system, Use-After-Cleanup (UAC) bugs can occur because a running kernel thread may be unaware of the device detachment and attempt to use an object that has been released by the cleanup thread. Our investigation suggests that an attacker can exploit the UAC bugs to obtain the capability of arbitrary code execution and privilege escalation, which receives little attention from the community. While existing tools mainly focus on well-known concurrency bugs like data race, few target UAC bugs.**

In this paper, we propose a tool named UACatcher to systematically detect UAC bugs. UACatcher consists of three main phases. It first scans the entire kernel to find target layers. Next, it adopts the context- and flow-sensitive inter-procedural analysis and the points-to analysis to locate possible free (deallocation) sites in the bottom-up cleanup thread and use (dereference) sites in the top-down kernel thread that can cause UAC bugs. Then, UACatcher uses the routine switch point algorithm which counts on the synchronizations and path constraints to detect UAC bugs among these sites and estimate exploitable ones. For exploitable bugs, we leverage the pseudoterminal-based device emulation technique to develop practical exploits.

We have implemented a prototype of UACatcher and evaluated it on 5.11 Linux kernel. As a result, our tool successfully detected 346 UAC bugs, which were reported to the community (277 have been confirmed and fixed and 15 CVEs have been assigned). Additionally, 13 bugs are exploitable, which can be used to develop working exploits that gain the arbitrary code execution primitive in kernel space and achieve the privilege escalation. Finally, we discuss UACatcher's limitations and propose possible solutions to fix and prevent UAC bugs.

I. Introduction

Device cleanup is triggered when a device is removed from the system. However, if the synchronization is not properly implemented in the OS kernel, a running kernel thread may be unaware of the device detachment and use an object that has been released by the cleanup thread, resulting in Use-After-Free (UAF) bugs [1,2]. What's worse, the device detachment can be issued from a user space program (without the requirement that the physical device is actually being removed), making the exploitation of this bug practical. This causes serious consequences to the whole system. Since this Concurrency UAF (CUAF) occurs *between the device cleanup thread and the running kernel thread*, it is named **Use-After-Cleanup (UAC)** in this paper. Though the UAC vulnerability can be exploited to launch the privilege escalation attack, to the best of our knowledge, there do not exist off-the-shelf tools that can systematically detect UAC bugs due to the following two reasons.

First, how to effectively detect Concurrency UAF bugs in *OS kernel* is an ongoing research effort in the community. The main focus of most dynamic fuzzing tools [3–7] is to find memory corruptions and they don't apply effective oracles to capture concurrency bugs. Besides, other systems [8,9], including Razzer, enhance the fuzzer with tailored algorithms to detect concurrency bugs. However, they mainly target the data race bug, which, in contrast to the UAC bug, is another different concurrency issue (as shown in Section II-D). Recently, Bai et al. propose DCUAF [10] that focuses on detecting CUAF bugs in Linux device drivers instead of data races. Nevertheless, the lockset analysis used in the paper is insufficient to detect UAC bugs since it introduces unnecessary false positives (as shown in Section IV-C).

Second, the lack of the detection tool is partially due to the neglect of the UAC bugs by the community *since it was improperly treated as a low-security impact*. For one thing, compared with direct memory corruption, the UAC bug as a concurrency issue is of less interest to attackers for its uncertainty. For another, in the community's thought, a UAC bug is triggered by malicious device detachment that requires a real and programmable device, which is expensive and unacceptable in a real attack scenario. Even though there are some virtual devices, e.g., the `vhci` Bluetooth controller, that can be detached from the system to trigger the cleanup in the kernel, these virtual devices require the root privilege, which downgrades the security impact. *However, our evaluation shows that these observations are not true, since there are UAC bugs that can be stably exploited from a user space program without the requirement of a real device nor the root privilege (as shown in Section V).* Due to the serious impact that the UAC bugs could cause, there is a pressing need to propose an efficient detection tool for such bugs.

Our Approach To this end, we propose UACatcher, the first tool that systematically detects UAC bugs in Linux kernel. Our tool is based on insight into the root cause of the UAC bugs. As shown in Fig. 1, a UAC bug is caused by the concurrent access from the kernel thread (❸) that is triggered by a system call (top-down access) [1], to the kernel object that has been

§Corresponding author

[1]Technically, the thread accessing the kernel object can also be triggered from the bottom-up access. However, in this work, we focus on the top-down access from the system call since the user space program can take this method to practically exploit the UAC bugs.

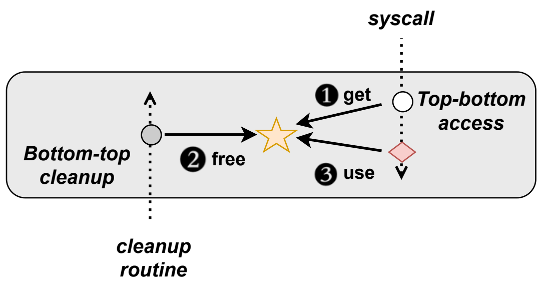


Fig. 1: The root cause of the UAC bug. The kernel object is freed by the bottom-top cleanup thread (❷) and used by the top-bottom access thread (❸).

released by the cleanup thread (❷) which is triggered from the underlying device detachment (bottom-up cleanup).

UACatcher consists of three main phases to detect UAC bugs. It first scans the entire kernel to find target layers and collects essential information about both cleanup routines and the syscall routines for these layers. It then adopts the context- and flow-sensitive inter-procedural analysis and the points-to analysis to locate possible free (deallocation) sites in the cleanup routine and use (dereference) sites in the syscall routine. After that, it detects the UAC bugs among these sites by analyzing the synchronizations and path constraints. For detected bugs, it outputs a report for us to manually confirm the true positives and prepare bug fixes. Moreover, UACatcher also helps identify the race windows for the detected UAC bugs. With the race windows, we estimate the exploitability of a UAC bug and further leverage the pseudoterminal-based device emulation technique to develop practical exploits.

We implement the tool prototype based on the CodeQL analysis engine [11]. The tool is evaluated on the 5.11 Linux kernel with 1,670 target layers. It finds 436 bugs. We manually check these bugs and find that 346 bugs are real. All the found bugs are reported to the Linux community and 277 of them have been confirmed and fixed and 15 CVEs have been assigned by the time of writing this paper (July 2022).

Additionally, we find 13 **exploitable** UAC bugs. We use them to build practical exploits that can gain the arbitrary code execution primitive and perform the privilege escalation. The video demonstrating the exploit is in the following link: https://github.com/uacatcher/uacatcher-repo/blob/main/demo.gif.

Contributions Our main contributions are in the following.

- We first model the Use-After-Cleanup, a special type of concurrent Use-After-Free bugs associated with the device cleanup routine and the syscall routine.
- We propose UACatcher, the first tool that fills the gap in UAC bug detection. It adopts the static analysis approach to detect UAC bugs from possible deallocation and dereference sites.
- We evaluate the UACatcher with 1,670 target layers of the Linux 5.11 (git commit 7289e26f395b) and find 346 true UAC bugs. All the detected bugs have been reported to the community and 277 of them have been confirmed and fixed and 15 CVEs have been assigned so far. Importantly, 13 bugs are exploitable. We then develop practical exploits based on them.
- We propose the possible solutions to fix and prevent UAC bugs according to the experience of submitting kernel patches and the communication with kernel maintainers.

```
struct device_driver {
    ....
    // Called when the device is removed
    // from the system to unbind a device
    // from its driver
    int (*remove)  (struct device *dev);
    ....
}

struct usbdrv_wrap {
    struct device_driver driver;
    ....
}

struct usb_driver {
    ....
    // Called when the interface is no
    // longer accessible, usually because
    // its device has been (or is being)
    // disconnected or the driver module
    // is being unloaded.
    void (*disconnect) (...);
    ....
    // embed device_driver
    struct usbdrv_wrap drvwrap;
    ....
}
```

Fig. 2: Structs and methods for handling device removal, USB host domain as an example

II. Background

A. Linux Device Removal Handling

To add or remove a device from the system bus dynamically, Linux kernel has integrated the hotplugging feature with its driver model core since the 2.4 version. The base device driver struct `device_driver`, as shown in Fig. 2, maintains a specialized function pointer `remove`, which is called when the device detaches from the host. On this basis, other derived device drivers, which inherit the base driver struct, maintain another customized function pointer to achieve driver-specific device removal handling. For example, the USB (host) subsystem, as also shown in Fig. 2, prepares the `usb_driver` struct. This struct has a member with the `usbdrv_wrap` type, which is used to inherit the `device_driver`. When a USB dongle is removed from the host, the function that is responsible for cleanup USB relevant resources is called via the member `disconnect` pointer.

B. Use-After-Cleanup Layered Model

When a device detaches from the system, the kernel will capture the interrupt signal and start the *device cleanup routine*. This routine will go through the kernel from bottom to top to notify each subsystem and layer about the device detachment. Besides, there is a routine known as the *syscall routine* that is triggered by the system call from a user space program. Contrary to the device cleanup routine, it will go through the kernel from top to bottom. As shown in Fig. 1, since the device cleanup routine is concurrent with the syscall routine, if the synchronization is not implemented properly, the top-down syscall routine may be unaware that the bottom-up

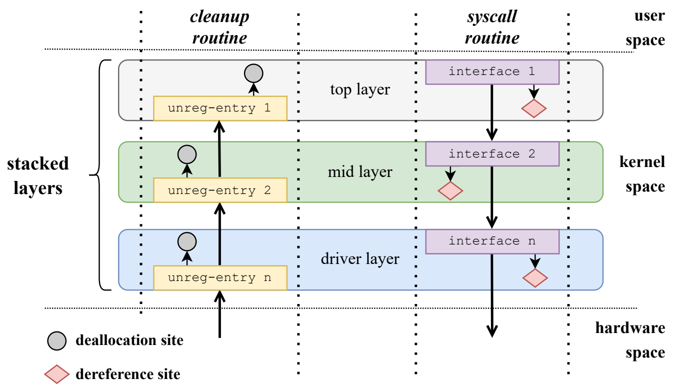


Fig. 3: Simplified sketch of UAC Layered Model. The routine that starts from the driver layer and handles the device detachment is named the device cleanup routine and the routine that starts from the top layer and performs the task of a system call is named the syscall routine. Code snippets in the device cleanup routine that free the resource objects are called deallocation sites and code snippets in the syscall routine that use the resource objects are called dereference sites. The unreg-entry and interface are boundary functions that break routines into subroutines.

cleanup routine is running, and use a resource object that has been already released, resulting in Use-After-Free (UAF) bugs. Since this kind of UAF is associated with the device cleanup routine and the syscall routine, we call it **Use-After-Cleanup (UAC)**. To describe the UAC bugs formally, we introduce the UAC Layered Model as presented in Fig. 3. The **deallocation site** is defined as the position where the bottom-up cleanup routine frees a kernel resource object. And the **dereference site** is defined as the position where the top-down syscall routine uses this kernel object. For clarity, we name a pair of a deallocation site and a dereference site for the identical kernel object as **dPair**. If the dereference site of a dPair is able to be scheduled after the corresponding deallocation site, this dPair causes a real UAC bug. Since the Linux kernel is organized by layers and the cleanup/syscall routine can be divided into layer-based segments, we refine the model by introducing the concept of the **layer-boundary functions**, which include the **unreg-entry** functions and the **interface** functions. As Fig. 3 shows, the unreg-entry function is defined as the entry point function of the device cleanup routine that a layer exposes to the lower layer. The interface function, in turn, is exposed to the upper layer for undertaking the syscall routine. With the boundary functions, we can break the entire device cleanup routine and syscall routine into several subroutines. Therefore, we are able to analyze these two concurrent routines layer by layer instead of the entire kernel, which offers better granularity for static analysis.

C. Use-After-Cleanup Example

We will present a real UAC bug in the Bluetooth stack. This bug was introduced in Linux 2.6.22-rc2 (May. 2007). It was fixed 14 years later (May, 2021) with the help of UACatcher. As Fig. 4 shows, the function `hci_unregister_dev` in

```
FILE: linux-5.11/net/Bluetooth/hci_core.c
3852. void hci_unregister_dev(…) {
      ……
3886.  hci_sock_dev_event(hdev, …); // ①
      ……
3899.  destroy_workqueue(hdev->workqueue);//②deallocation
3900.  destroy_workqueue(hdev->req_workqueue);
      ……
3922. }
```

```
FILE: linux-5.11/net/Bluetooth/hci_sock.c
 734. void hci_sock_dev_event(…) {
      ……
 767.   hci_pi(sk)->hdev = NULL;
      ……
 778. }
------------------------------------------------------------
1701. static int hci_sock_sendmsg(…) {
      ……
1746.  hdev = hci_pi(sk)->hdev; // ③
      ……
1829.  queue_work(hdev->workqueue, …); // ④ dereference
      ……
1841. }
```

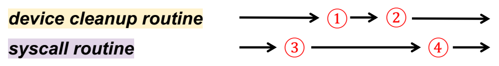


Fig. 4: A reported UAC bug in the Bluetooth stack and the routine interleaving sequence for triggering it.

`hci_core.c`, which is the unreg-entry of the HCI top layer, will call the `destroy_workqueue` on line 3899 after it notifies all working sockets with function `hci_sock_dev_event` to reclaim the target resource object `hdev->workqueue`. In `hci_sock.c`, the function `hci_sock_sendmsg` needs to use this resource object on line 1829 when a `sendmsg` system call is issued. Therefore, the line 3899 site and line 1829 site constitute an interested dPair. When the device cleanup routine and the syscall routine are interleaved as Fig. 4 presents, the deallocation site of this dPair ② is scheduled before the dereference site ④. In other words, the function `destory_workqueue` will deallocate the memory of the resource object, and thus the function `queue_work` will dereference a dangling pointer, leading to a UAC bug.

D. Data Races and UAC Bugs

A data race occurs when two threads are accessing one shared memory location, at least one of the two accesses is a write, and the two accesses are not synchronized properly [12–14]. Many previous works leverage static [15,16] or dynamic [17,18] analysis techniques to reveal such bugs in the kernel, especially in device drivers [12,19,20] and file systems [8,9]. However, the Use-After-Cleanup (UAC) bug proposed in this work is a special type of CUAF bug that is different. Specifically, different from the definition of data race bugs, CUAF bugs are concurrency bugs that only occur when the dereference of a dynamic object can be scheduled after the deallocation due to synchronization issues. For UAC bugs we target, the deallocation is associated with the device cleanup routine and the dereference is associated with the syscall routine. To detect a CUAF bug, we cannot directly leverage the data race detector because of one primary reason: If we choose data race detection as a springboard to detect CUAF, it is non-trivial to determine if the detected data race actually leads to a CUAF as not all data races are harmful in Linux kernel and it is hard to decide on that [9,21]. For instance, Bai et al. detect 149K data races in Linux 4.19 drivers while many of them are benign [10]. To this end, they choose to detect the CUAF in the Linux device drivers by remodeling the bug and leveraging the lockset algorithm [10]. Their implemented tool DCUAF is the state-of-the-art in detecting CUAF in the kernel which successfully detects hundreds of bugs. Nevertheless, we find that this solution is insufficient to detect UAC bugs since it introduces unnecessary false positives (as shown in Section III and Section IV-C).In summary, even with mature detectors for data races and other concurrency bugs, there is still a lack of effective approaches to accurately detect UAC bugs.

III. Key Challenges

Detecting UAC bugs in the kernel is non-trivial due to the three following challenges.

C-I: How to prepare target layers? There are a huge amount of layers in the entire linux kernel. However, not all of them are vulnerable to UAC bugs. Since it is difficult to analyze all these layers, we need to determine which of them is an appropriate target that is more likely to be the victim of the UAC bug. Additionally, for a target layer, how to find its boundary functions (unreg-entry function and interface functions) remains another challenging task.

C-II: How to accurately locate dPairs? We note that it is challenging to accurately locate an expected dPair. That is because, given a dPair located by a static analysis approach, the corresponding object which is dereferenced and the object being released is not necessarily the same one [22]. Erroneously located dPair causes false positives. For example, the DCUAF [10] has reported an unreal bug[2] because the dPairs are not accurately located.

C-III: How to accurately detect UAC bugs? As we discuss ahead, there is a lack of effective approaches to accurately detect UAC bugs. The methods proposed by the existing works cannot be ported to detect UAC bugs accurately. Some of them [23–25] mostly focused on user space multithreaded programs hence not scalable enough to target the kernel. The others [10,20] propose the lockset analysis method which introduces unnecessary false positives (as shown in Section IV-C).

IV. UACatcher Design

A. Layers Preparing

1) Finding Stacked Layers via Cleanup Routine

Because not all layers in the kernel are vulnerable to UAC bugs, i.e., layers (synchronization, security, etc) that are almost independent of device domains, it is necessary for UACatcher to find proper layers first otherwise effort will be wasted on non-interesting targets. Additionally, found layers, compared with the entire kernel, are fine-grained targets that can improve the efficiency and accuracy of static analysis.

UACatcher's solution for this is to find **stacked layers**. As shown in Fig. 3, stacked layers are layers that are stacked on top of each other where the bottom is the driver layer. In another word, layers within stacked layers are interconnected via the unreg-entry functions and interface functions discussed in Section II-B. To find stacked layers in the kernel, we first find the bottom driver layers and then find the upper layers. Specifically, UACatcher leverages the device driver model knowledge and cleanup routine knowledge mentioned in Section II. It first inspects the entire kernel to find all derived driver types by checking out if a driver structure embeds `device_driver` as a member. After that, we manually mark the pointer field responsible for device removal for all found driver structures. For instance, we mark the `disconnect` field in structure `usb_driver` as demonstrated in Section II-A. Note that this is accurate and efficient as the comment of the driver structure clearly explains the usage of all pointer fields. On that basis, UACatcher then finds all driver objects with type analysis and extracts the unreg-entry functions through the marked pointer fields. After the unreg-entry function of a layer is pinpointed, UACatcher then gets the whole picture of this layer by parsing the *Kbuild* and *Makefile* code. Based on that, UACatcher starts to find upper layers that are stacked above the driver layers according to two insights: (1) the unreg-entry function of the upper layer must be called once from the lower layers because the cleanup routine goes through layers from the bottom to top; (2) the unreg-entry function of the upper layer has the following characteristics according to our practice of going through the source code.

- **Type characteristic:** the unreg-entry functions are void (nonValue-returning) functions.
- **Parameter characteristics:** the unreg-entry functions have only one pointer parameter.
- **Name characteristics:** the name of unreg-entry functions contains tacit keywords, i.e., `unregister`.

For instance, the function `void hci_unregister_dev(struct hci_dev *hdev)` is used in the top layer for the Bluetooth stack and is called in Bluetooth driver layers. And function `void nfc_unregister_device(struct nfc_dev *dev)` is used in the core layer for the NFC network stack.

Hence, UACatcher performs cross-reference analysis from already found driver layers to find upper layers and recursively uses new-found upper layers as input to find new layers. Eventually, we obtain all stacked layers covered in the cleanup routine from the entire kernel with their unreg-entry functions pinpointed.

2) Gathering Interface Functions for Syscall Routine

According to the layered model presented in Fig. 3, the interface functions are exposed by the lower layer to the upper layer for undertaking the top-down syscall routine. UACatcher gathers these functions via two steps: (1) Find the potential interface functions as many as possible. As is known, the interface function is always called via a function pointer

[2]kernel commit 3f60f468569 ("cw1200: Revert unnecessary patches that fix unreal use-after-free bugs")

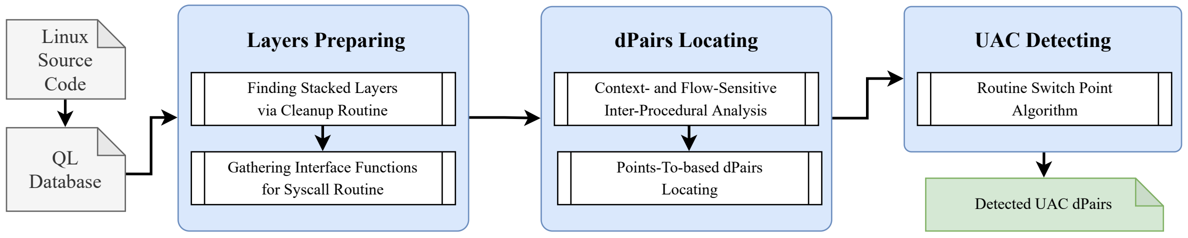

Linux Source Code
QL Database
Layers Preparing
Finding Stacked Layers via Cleanup Routine
Gathering Interface Functions for Syscall Routine
dPairs Locating
Context- and Flow-Sensitive Inter-Procedural Analysis
Points-To-based dPairs Locating
UAC Detecting
Routine Switch Point Algorithm
Detected UAC dPairs

Fig. 5: The overview of UACatcher. Its detection consists of three main phases. In the first phase, it scans the entire kernel to find a number of target layers and collects essential information, such as the boundary functions for these layers. Then it locates the dPairs that may cause UAC bugs via context- and flow-sensitive inter-procedural analysis and the points-to analysis. It finally detects the UAC bug among these dPairs with the routine switch point algorithm and outputs the detected results. The detected UAC bugs will be further confirmed with manual efforts.

[26,27]. Moreover, similar to how an unreg-entry function of a driver layer is packed, interface functions are always packed together in a specific (static, global) structure. For example, all interface functions of a network layer are packed into `proto_ops` structure. Unfortunately, because there are too many such structures and there is no unified pattern among them, it is non-trivial to manually find all of these structures. Therefore, UACatcher looks for all possible structures and extracts their initialized function pointer members to avoid false negatives. (2) Confirm interface functions from call sites in the upper layer. To reduce the false positives, UACatcher filters the found interface functions of a layer by confirming whether the upper layer accesses it. Specifically, for an interface function extracted from member `A` within the structure whose type is `B`, UACatcher iterates all call sites in the upper layer to find if there is an indirect call that dereferences a structure pointer with type `B` and accesses member `A`. If so, UACatcher considers this interface function as a true positive. Otherwise, UACatcher discards this function as it is never called from the upper layer.

B. dPairs Locating

1) Context- and Flow-Sensitive Inter-Procedural Analysis

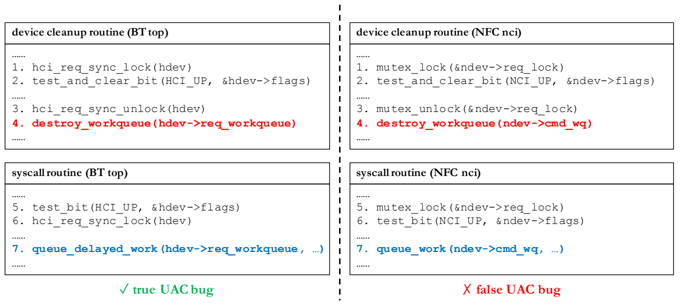
UACatcher adopts context- and flow-sensitive inter-procedural analysis to achieve accurate static analysis. For an input layer, we generate its (directed acyclic) call graph by traversing the call relations starting from the boundary functions. Specifically, we initialize an analysis stack with an entry function and loop the analysis until the stack is empty. When analyzing the function that popped from the analysis stack in one loop, we break it into basic blocks and construct its control-flow graph. Hence, all function calls inside the analyzed function can be determined from the CFG in a flow-sensitive manner. If the called function is defined in other layers, it will be regarded as an external symbol and the loop continues. Other called functions, together with the call sites information and passing arguments will be pushed into the analysis stack for later processing. UACatcher outputs a call graph after the entire traversal. The nodes of this graph are actually control-flow graphs of the corresponding functions and the edges of this graph consist of detailed call sites and arguments. Once the graph is constructed, it is used in points-to analysis and UAC bug detection.

2) Points-To-based dPairs Locating

deallocation site. During the generation of the above call graph from the unreg-entry function, we pinpoint the deallocation sites by tracking down the call sites to the deallocation functions, `kfree()` as an example. To accomplish this, UACatcher collects the most commonly used deallocation functions, such as `kfree()`, `kmem_cache_free()`, that are disclosed in previous works [10,22,28]. Additionally, UACatcher combines static analysis to find other deallocation functions for specialized objects. These functions always receive only one specific pointer of the object as the argument hence we can use signature-based analysis to find possible candidates. As we find that most deallocation functions have keywords like `destroy`, `free`, `release` in their names. UACatcher then extracts real deallocation functions from the candidates by regex matching and manual confirmation. For example, deallocation functions like `destroy_workqueue()`, and `kfree_skb()` are found.

dereference site. UACatcher leverages field-sensitive refined points-to analysis to locate the relevant dereference sites. Besides all pointer dereference expressions, if a pointer is passed to an external function as a parameter, we also regard it as a dereference action as we have no idea how this function operates the pointer. During the analysis, UACatcher iterates through all dereference actions of a layer and checks if the action, either expression or parameter, points to the same location as one of the previously pinpointed deallocation sites does. If so, UACatcher marks this action as the dereference sites for the corresponding deallocation site. To sum up, UACatcher adopts a summary based context- and flow-sensitive inter-procedural analysis and points-to analysis to locate the possible deallocation sites and the dereference sites.

paths. For all located deallocation sites and their corresponding dereference sites, UACatcher constructs all the simple paths whose sources are the boundary functions and sinks are these sites. The deallocation paths start from the unreg-entry function and end at the deallocation sites while the corresponding dereference paths start from one of the interface functions and end at the corresponding dereference sites. Note that the paths are not just a list of function nodes. They also take the control-flow graph into account. If there are several call sites that allow parent function to call child function,



```
device cleanup routine (BT top)
......
1. hci_req_sync_lock(hdev)
2. test_and_clear_bit(HCI_UP, &hdev->flags)
......
3. hci_req_sync_unlock(hdev)
4. destroy_workqueue(hdev->req_workqueue)
......

syscall routine (BT top)
......
5. test_bit(HCI_UP, &hdev->flags)
6. hci_req_sync_lock(hdev)
......
7. queue_delayed_work(hdev->req_workqueue, …)
......
```

✓ true UAC bug

```
device cleanup routine (NFC nci)
......
1. mutex_lock(&ndev->req_lock)
2. test_and_clear_bit(NCI_UP, &ndev->flags)
......
3. mutex_unlock(&ndev->req_lock)
4. destroy_workqueue(ndev->cmd_wq)
......

syscall routine (NFC nci)
......
5. mutex_lock(&ndev->req_lock)
6. test_bit(NCI_UP, &ndev->flags)
......
7. queue_work(ndev->cmd_wq, …)
......
```

✗ false UAC bug

Fig. 6: Examples of a true UAC bug in the Bluetooth stack (left) and a false one in the NFC stack (right). The state-of-the-art tool reports the right one and causes false positives.

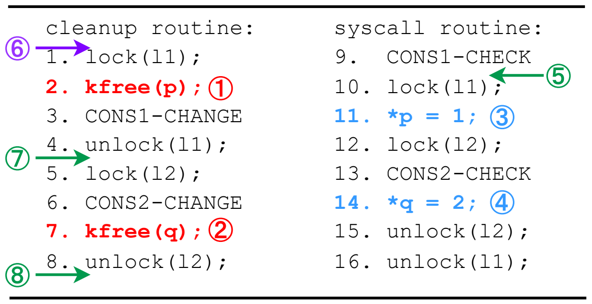
UACatcher will record all possible basic blocks paths to promise a sound analysis result.

Moreover, to reduce the false results, UACatcher adds several filters: (1) We set the threshold value for points-to analysis. That is, if the points-to analysis fails to accurately pinpoint the point sets (which is common in the kernel analysis case) and gets low confidence, we discard the relevant site. (2) The filter checks whether the located deallocation site is within the same function as the allocation site. If so, this deallocation site is discarded as it most likely acts as error handling which has little relation to UAC bugs. (3) For each pair of the deallocation path and the dereference path, the filter collects and checks the sets of their common segments. If two paths have common segments, we drop this pair of paths as these paths may actually not be executed concurrently [10]. One dPair will be discarded if there are no paths remaining to reach it.

C. Use-After-Cleanup Detecting

In this phase, UACatcher detects if the located dPairs lead to real UAC bugs. To illustrate the difference between a real UAC bug and a false one, an example is given in Fig. 6. The path in the left one causes a true UAC bug, of which the target object is `req_workqueue`. The reason why this bug occurs is that the syscall routine places the check of the `test_bit(HCI_UP)` out of the critical section held by the `hci_req_sync_lock`. That is, there is a possibility that the syscall routine passes the `test_bit` first and then waits on the `hci_req_sync_lock` because the device cleanup routine just holds this lock. In another word, the read and write to the `hdev->flags` is not well protected from a race condition. Hence, the deallocation site (`destroy_workqueue`) is able to be reached before the dereference site (`queue_delayed_work`), causing a UAC bug. The path in the right one, however, leads to no UAC bug. The found dereference site can never be reached after the deallocation site is executed. That is because the read and write to the `ndev->flags` are protected by the lock `req_lock`. After the deallocation site is executed, the `NCI_UP` flag is cleared out hence the `test_bit` constraint check before the dereference site `queue_work` is impossible to be reached.

We note that the lockset analysis, which is used by the state-of-the-art works [10], brings out false results in detecting UAC bugs. For the false UAC bug in Fig. 6, because the



```
     cleanup routine:          syscall routine:
⑥ → 1. lock(l1);              9.  CONS1-CHECK
     2. kfree(p); ①           10. lock(l1); ← ⑤
     3. CONS1-CHANGE           11. *p = 1; ③
⑦ → 4. unlock(l1);            12. lock(l2);
     5. lock(l2);              13. CONS2-CHECK
     6. CONS2-CHANGE           14. *q = 2; ④
     7. kfree(q); ②           15. unlock(l2);
⑧ → 8. unlock(l2);            16. unlock(l1);
```

Fig. 7: Example traces of the device cleanup routine and the syscall routine used for explaining the Algorithm 1. red represents the deallocation sites; blue represents the dereference sites; purple represents the start of the device cleanup routine; green represents the context switch points. The CONS-CHECK/CONS-CHANGE are abbreviations for constraint check and constraint change, respectively.

intersection of the dereference lockset ({req_lock}) and the deallocation lockset ($\emptyset$) is empty, the lockset analysis will recognize it as a true bug, causing a false positive. Hence, to accurately detect the UAC bugs, UACatcher should check the happen-before relation between the dereference site and the deallocation site. Inspired by the previous work [29], we propose the *routine switch point* algorithm to infer the happen-before relation. The core idea of this algorithm is trying to find appropriate context switch points which allow the dereference site to be scheduled after the deallocation site based on the synchronizations and the path constraints. If no context switch point is found, the dereference site hence cannot be scheduled after the deallocation site, which means that the happen-before relation holds between the dereference (use) and the deallocation (free).

Algorithm 1 defines how UACatcher detects a UAC bug. We first explain some abbreviated macros before looking into the routine switch point algorithm. The `HHoldLock` gets the historical holding locks for an input position; The `CHoldLock` gets the current holding locks, or lockset, for an input position; The `Pred` gets the previous position of the input position of an input path; The `Succ` gets the next position of the input position of an input path; Upon that, the main function `UACDetect` receives the deallocation path $Path_1$ and the dereference path $Path_2$ of a given dPair as input. If this function returns $True$, the dPair is considered to cause a UAC bug. In addition, the function outputs the context switch points P which can help to analyze and reproduce the bug. This algorithm first collects the constraint checks with reverse execution (line 4-9) until the intersection between the `HHoldLock` of the deallocation site and the `CHoldLock` of the executing site is empty. When this while loop is over, the position where executing site stops is regarded as the first context switch point (switch-A). The execution is expected to be switched from the syscall routine to the cleanup routine at switch-A. Because the result of `InterLockA` is empty, the switched execution is able to run from the unreg-entry till the cleanup routine releases the

Algorithm 1: Routine Switch Point Algorithm

```
Input  : deallocation path $Path_1$, dereference path $Path_2$
Output: context switch points $P$
1  Function UACDetect()
2      $S_1 = \texttt{End}(Path_1), S_2 = \texttt{End}(Path_2)$
3      $r = S_2, PC1 = \emptyset, PC2 = \emptyset$
4      $L_{inter} = \texttt{InterLockA}(S_1, r)$
5      while $L_{inter} \neq \emptyset$ do
6          $r = \texttt{Pred}(r, Path_2)$
7          if $\texttt{Action}(r) \in$ ConstraintCheck then
8              $PC1 = PC1 \cup \{\texttt{Action}(r)\}$
9          $L_{inter} = \texttt{InterLockA}(S_1, r)$
10     $P = P \cup \{r\}$
11     $r = \texttt{Start}(Path_1)$
12     $L_{inter} = \texttt{InterLockB}(S_2, r)$
13     while $L_{inter} \neq \emptyset$ do
14         $r = \texttt{Succ}(r, Path_1)$
15         if $\texttt{Action}(r) \in$ ConstraintChange then
16             $PC2 = PC2 \cup \{\texttt{Action}(r)\}$
17         $L_{inter} = \texttt{InterLockB}(S_2, r)$
18     $P = P \cup \{r\}$
19     if Satisfy $(PC1, PC2)$ then
20         return $True$
21     return $False$
22 Function InterLockA($s1, s2$)
23     return $\texttt{HHoldLock}(s1) \cap \texttt{CHoldLock}(s2)$
24 Function InterLockB($s1, s2$)
25     return $\texttt{CHoldLock}(s1) \cap \texttt{CHoldLock}(s2)$
```

acquired locks that hinder the syscall routine from reaching the dereference site. (line 12-17). During this execution, all the constraint changes are collected. When this while loop is over, the stopping position is regarded as the second context switch point (switch-B), where the execution can switch back to the syscall routine and try to reach the dereference site [3]. Finally, this algorithm checks the satisfaction of the collected path constraints changes & checks. If an input dPair passes this check, which means the dereference site of this dPair is able to be reached after the context switch at switch-B, a UAC bug is detected. The context switch points (switch-A and switch-B) that cause the UAC are named the routine switch points.

Fig. 7 shows simple example traces that are used to illustrate how this algorithm works. The traces contain two dPairs: (1) dPair-1, deallocation site ① and dereference site ③. (2) dPair-2, deallocation site ② and dereference site ④. For the dPair-1, the intersected lock L_{inter} is initialized with the `InterLockA` of the deallocation site ① and the dereference site ③, which is {l1} (line-2). In the first while loop, no constraint check is collected and the reverse execution stops at ⑤ where the syscall routine has not yet obtained the lock `l1`. Thus, ⑤ is regarded as switch-A, and the execution switches to the start of the device cleanup routine marked as ⑥. In the second while loop. one constraint change is collected in the $PC2$ ({CONS1-CHANGE}), and the execution stops at ⑦, that is, the switch-B. Because the collected constraint checks $PC1$ (empty) are not contradictory to the collected constraint changes $PC2$ ({CONS1-CHANGE}), dPair-1 is considered to cause a UAC bug. Additionally, the dPair-2 is detected to be a false UAC, because Algorithm 1 will find that the $PC1$ ({CONS2-CHECK}) and the $PC2$ ({CONS1-CHANGE, CONS2-CHANGE}) are contradictory to each other. In another word, after switch-B, the syscall routine can never reach the dereference site and cause UAC bugs.

[3]Switch-B can happen before the deallocation site. In that case, the subsequent deallocation and dereference may happen concurrently, which means UAF is possible.

To implement the Algorithm 1 for UACatcher, we model synchronizations with commonly used locks in Linux kernel, including mutex, spinlock, read/write locks, and customized wrappers of them [18]. UACatcher analyzes all exported symbols within `kernel/locking` to get these lock/unlock functions. For path constraints, since there are sundry types of path constraints in the kernel code and different protocol stacks maintain dissimilar path constraints such as flags of states, it is challenging to accurately extract the checks and the changes. Currently, UACatcher supports two typical constraints: (1) pointer nullification and relevant check; (2) constraints that operate on bits, the `test_bit` for instance. In our practice, we found that most layers count on these two constraints to maintain the state machine. The possible false alarms caused by the incomplete support of synchronizations and path constraints will be discussed in Section VII.

V. UAC Bug Estimation

UAC bugs could lead to serious security impacts like privilege escalation. In this section, we estimate the UAC bug by taking two key points into account: (1) If this UAC bug can be easily triggered? (2) If this UAC bug can be exploited without real devices?

A. Race Window Identification and Measurement

A UAC bug happens with uncertainty. To assess the bug triggering possibility, one has to analyze its root cause and identify its race windows, which is challenging. Luckily, as the routine switch point algorithm has already pinpointed the context switch points, we can accurately identify the race windows of a UAC bug. To accomplish that, UACatcher continues the reverse execution from the located switch-A and stops the execution when it finds an additional constraint check that is contradictory to the collected constraint changes. After that, the race window is identified as starting from the execution stop position and ending at switch-A. Taking the UAC bug from the deallocation site ① and dereference site ③ as an example, the reverse execution starts from the switch-A (⑤) and ends at the CONS1-CHECK statement as this constraint check conflicts with the collected ({CONS1-CHANGE}). Hence, the race window is from the CONS1-CHECK to lock(`l1`) (9-10). If the cleanup routine acquires the lock `l1` during this race window, the UAC bug is about to be triggered. Since the race window in this example is too small, UACatcher considers this UAC is hard to trigger.

To find UAC bugs that can be easily triggered, UACatcher measures the size of the race window to estimate the trig-

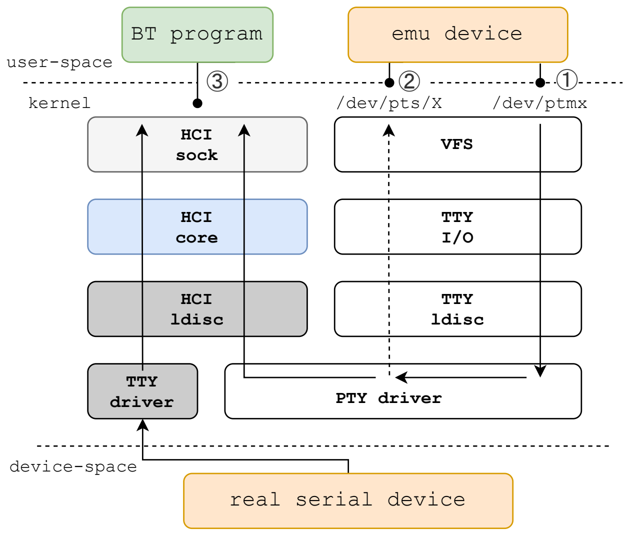


Fig. 8: Steps to emulate a Bluetooth device in user space with pseudoterminal. ①: open `/dev/ptmx` device file, this will generate a corresponding slave device in `/dev/pts` directory. ②: use `ioctl` to switch and register the generated slave device with `N_HCI` line discipline. ③: create an HCI socket and bring the device up. A real serial BT device is also shown to demonstrate the similarity.

gering possibility. In addition, during the reverse execution, UACatcher checks if there are any *time-consuming* functions or *time-controllable* functions being called within the race window. Specifically, the time-consuming functions include memory allocations/deallocations, logging, and I/O operations that cost numbers of CPU cycles thus enlarging the race window. The time-controllable functions are mainly helper functions that exchange data between kernel space and user space, e.g., the `copy_{from/to}_user`. If these functions are called within the race window, the attacker can leverage techniques like `userfaultfd` and `fuse` to fully manipulate the consumed time [30,31]. Once time-consuming functions or time-controllable functions are found, the UAC bug is considered to be easily triggered.

B. User Space Device Emulation

To exploit a UAC bug that is estimated to be easily triggered, UACatcher leverages the user space device emulation technique [32]. We'll resort to two primary building blocks below to introduce this technique.

Pseudoterminal Device. A pseudo-terminal (sometimes abbreviated "pty") [33] is a pair of virtual character devices endpoints (files) that provide a bidirectional communication channel. One end of the channel is called the *master*; the other end is called the *slave*. In general, the master endpoint files are typically used by networking applications like *ssh* while the slave files are used by terminal-oriented programs such as shells like *bash* [34]. On Linux systems, a user program A can open `/dev/ptmx` device file to obtain the descriptor of the master endpoint. At the same time, the PTY driver in the kernel will allocate a unique slave endpoint file under `/dev/pts/` directory that emulates a hardware text terminal device which supports System V API. [4] So far, the slave device is ready to be opened by other processes that want to establish the IPC channel with program A.

Line Discipline. The job of the TTY device driver in the kernel is to format data that is sent to it in a manner that the hardware can understand, and receive data from the hardware [35]. To control the flow of data, there are a number of different line disciplines that can be selected and virtually "plugged" into any TTY device. The TTY line discipline (ldisc)'s job is to format the data received from a user, or the hardware, in a specific manner. For instance, for a serial-port-based Bluetooth controller, to format the data according to the form of the HCI protocol conversion, a user has to open and set the HCI line discipline for this device. Only then can this controller be identified as a Bluetooth HCI dongle by the kernel rather than other hardware devices.

Fig. 8 shows the steps to emulate a Bluetooth (BT) device. First, the attacker opens the `/dev/ptmx` to create pseudoterminal devices and obtain the file descriptors `f1`, `f2` of the master and slave endpoints. After that, the attacker uses `ioctl` system call for `fd2` to register the BT line discipline (`N_HCI`) for the slave device, which is default registered with the TTY line discipline (`N_TTY`). After the line discipline is switched, the attacker has to prepare program logic that handles the data transaction when attaching a specific BT controller. That is, once this emulated device is brought up, the attacker needs to read data requests from `fd1` and write replying data packets to `fd1`. To awaken the device cleanup routine, the attacker just needs to close the `fd2` and the PTY driver will kill the emulated device and send the signal to the line discipline layer.

We note that there are distinct differences between the pseudoterminal-based emulated device and the virtual device (e.g., the `vhci`) which is designed for testing and fuzzing purposes. Firstly, pseudoterminal-related code always exists in the kernel while the virtual device code needs to be configured and actively installed, which requires root privilege. Additionally, operations, such as `open` and `ioctl`, on these virtual devices also require root privilege hence raising less interest to attackers. In contrast, the pseudoterminal-based emulated device only asks for low privileges. According to our practice, we find that the BT stack asks for the `CAP_NET_ADMIN` when bringing up the device and the Amateur Radio and Controller Area Network stack asks for the `CAP_NET_ADMIN` when switching the line discipline. To our surprise, we find that the NFC stack requires no privilege for emulating a device. That is, permission checking is missing for operations like powering up the device or configuring the device[5]. This implies that pseudoterminal-based device emulation exposes an underestimated but potentially dangerous attack surface, thus improving the exploitability of the UAC bugs.

Recent work Frankenstein [36] also leverages pseudoterminal but its utilization is totally different from UACatcher.

[4]To support BSD API, `/dev/ptyX` is used for master devices and `/dev/ttyX` for slave devices.

[5]This is also reported and fixed now in upstream

To be specific, Frankenstein leverages pseudoterminal as a bridge to allow the user space fuzzer to talk to the operating system and handles the details by using the existing tool `btattach`. In comparison, UACatcher digs into the internal of the pseudoterminal and then implements flexible emulators that can integrate techniques like `userfaultfd` which is useful in developing practical exploits. Moreover, Frankenstein only focuses on utilizing pseudoterminal for the BT stack while UACatcher focuses on cases for all possible layers, such as the Amateur Radio and Controller Area Network stack, etc.

In summary, when a detected UAC bug is estimated to be easily triggered, we further check whether it can be triggered via user space device emulation. If so, we start the further auditing of this UAC to get details (e.g., the use-after-free read/write size and offset). The idea for exploiting a use-after-free vulnerability in kernel is straightforward: spray appropriate vulnerable objects to build attack primitives [22,37]. To highlight the estimation ability of the UACatcher, we introduce how it helps us to exploit the UAC bug shown in Fig 4. The identified race window for this UAC is from ③ to ④. During the estimation, UACatcher finds a time-consuming function: `sock_alloc_send_skb` and a time-controllable function: `memcpy_from_msg` to confirm the high triggering possibility. Based on that, with the pseudoterminal-based user space device emulation, our final exploit leverages the `userfaultfd` to issue controllable page fault during the race window and detaches the device in page fault handling to constitute a 100% successful attack (as shown in the demo video).

VI. Implementation

We implement the static analysis part of UACatcher with Python code and CodeQL v2.7.2 [38]. CodeQL is an open-source tool developed by GitHub. It is a semantic code analysis engine that allows the researchers to automate the security checks and perform variant analysis on software code based on languages such as JavaScript, TypeScript, Python, C, C++, C#, and Go. CodeQL is based on the QL [11] query language and provides the standard libraries, queries, extractors, and plugins [39] to support advanced static analysis techniques, like data flow analysis and taint analysis. Although the LLVM infrastructure [40] is a common choice for building a static analysis tool, UACatcher prefers CodeQL for its simplicity. For example, as CodeQL will convert the statements in kernel source code into classes that are able to be queried, so UACatcher can easily find all specific statements like pointer dereference statements.

Because there is no standard CodeQL library for the Linux kernel, we do a porting from the user space C/C++ library. To interact with the CodeQL engine from our Python code, we implement a driver module with batch query optimization. That is, for some similar queries, UACatcher will assemble them in one batch query to optimize the efficiency. Specifically, we write below types of CodeQL query in UAC bug detection: (1) queries that collect lock/unlock functions, deallocation functions, boundary functions, and other one-time effort information; (2) queries that are used to deduce calling relations for building callgraphs and flow relations for building control-flow graphs; (3) queries that leverage points-to techniques to find dereference sites from the deallocation site. Other tasks, like detecting with the routine switch point algorithm, are finished in our Python detector[6].

UACatcher is the first tool that detects the UAC bugs in Linux kernel. We hope that our work can motivate future research to focus on the possible vulnerabilities related to the device cleanup routine. For reproducibility concerns, the source code of the UACatcher and the program for device emulation are available at https://github.com/uacatcher/uacatcher-repo.

VII. Evaluation

This section presents the evaluation result of UACatcher. We first introduce the setup and configuration in the evaluation and then answer the following research questions.
RQ1: Is UACatcher able to find and detect UAC bugs?
RQ2: Is UACatcher able to estimate exploitable UAC bugs?
RQ3: Is UACatcher outperforming the state-of-the-art tools
RQ4: Is UACatcher portable to handle more targets?

A. Build CodeQL Database

Before leveraging CodeQL engine for static analysis, we need to build the database containing all the data required to run QL queries. In a nutshell, we should specify the building commands to invoke the required build system in the Linux kernel and the CodeQL extractor will extract necessary queryable data. To analyze the entire kernel, we choose the Linux kernel 5.11 (released in February 2021, git commit 7289e26f395b) and build the first database in allyesconfig. However, querying such a huge database is slow and inaccurate. As mentioned in Section IV, we choose layers as our interesting targets and build layer-granularity databases for bug detecting. To do that, UACatcher parses the Kbuild and Makefile to obtain a list of object files that constitute a layer and assemble the building commands based on these files.

B. Detect UAC Bugs (RQ1)

UACatcher detects UAC bugs via three phases. Thus, we will examine the outcomes of each phase in detail.

1) Layer preparing phase

As presented in Fig. 9, UACatcher finds 1,856 target layers extracted from 89 stacked layer components. Additionally, UACatcher successfully gathers interface functions for 1,670 of them. Specifically, within this phase, we find and mark 88 types as shown in Table I. To check if there are any missing types, we find all possible driver types based on regex matching and manually verify the difference set. As a result, no false negatives are found while the types in the difference set are mostly virtual drivers that do nothing with device removal. Based on these types, UACatcher finds 3,579 driver layers and gets 1,678 driver layers among them which act as bottom layers for 178 upper layers. Since UACatcher

[6]One can also implement the prototype only with CodeQL, we choose to use Python for flexibility.

Phase-1				
driver types	stacked layers	picked total layers		
88	89	1,678 driver + 178 upper (17 FP and 21 FN)		
Phase-2				
located dPairs		discarded dPairs		remaining dPairs
136,628		82,912 + 2,446		51,270
Phase-3				
detected dPairs		false alarms	true bugs	new bugs
436		90	346	337

Fig. 9: Evaluation results for each phase.

finds unreg-entry functions of upper layers with heuristics that could lead to false negatives and false positives, we examine the outcome via two steps. Firstly, we cluster the driver layers for which UACatcher fails to find upper layers and manually investigate them. Consequently, we find 21 false negatives. UACatcher fails to find them because they don't follow the naming rule as most upper layers do. For example, five false negatives found in NFC subsystems use *remove* as their keyword instead of *unregister*. To this end, we simply add these layers into the target layers pool. Secondly, we verify the 1,856 found layers and their pinpointed unreg-entry functions and find 17 false positives. The reason is that the naming rule also matches some functions used to unregister something else, such as notifiers (`unregister_memory_notifier`), which are out of our expectations. Afterwards, we input 1,860 true layers to UACatcher and get 1,670 output layers (89.8%) whose interface functions are successfully gathered. For 190 failed cases, we manually investigate them to find out the failure reasons. Consequently, we find that 81 of them are fine to be ignored because they register a dummy device (e.g., `i2c_new_dummy_device`) that requires no interface functions. Other failed cases are due to the **irregular layer structure** issue which will be discussed in Section VIII.

2) dPairs locating phase

To locate the deallocation sites, we collect 16 commonly used deallocation functions[7] as shown in Table II Thus, UACatcher successfully locates 1,155 deallocation sites for all target layers. Based on that, UACatcher then locates 136,628 dereference sites by leveraging points-to analysis. For all these located deallocations sites and their corresponding dereference sites, UACatcher constructs paths that start from boundary functions to reach those sites. With the help of these paths and three filters mentioned in Section IV, we discard 85,538 false dPairs (2,446 from the points-to filter and 82,912 from the other two filters). In summary, as presented by Fig. 9, UACatcher locates 51,270 dPairs from 315 layers. With manual investigation, we find that those layers for which UACatcher fails to locate dPairs mostly count on other resource deallocation mechanisms, e.g., RCU. Since those mechanisms are intrinsically safe against UAC bugs due to their carefully designed reference counting, we regard them as out-of-scope sites.

[7]Besides, there are more customized deallocation functions for dedicated objects. We regard them as future targets.

layer	description	impact
BT	Read connection information from a destroyed device	□
	Read authentication information from a destroyed device	□
	Add address to the blacklist of a destroyed device	◊
	Remove address from the blacklist of a destroyed device	◊
	Enqueue a work to a destroyed workqueue	⋆
	Fetch an already reclaimed connection object	⋆
	Fetch and dereference a NULL resource pointer	△
NFC	Enqueue a work to a destroyed workqueue	⋆
	Fetch and dereference a NULL resource pointer	△
AX25	Read address information from a destroyed device	□
	Copy controllable data into reclaimed buffer	⋆
	Fetch and dereference a NULL resource pointer	△
MCTP	Read route information from a destroyed device	□

Fig. 10: 13 exploitable bugs and their impact. □: can read particular data from a destroyed object; ◊: can write particular data to a destroyed object; △: can cause Denial of Service (DoS); ⋆: the destroyed object is viewed as a normal object to do complex tasks. All these bugs are reported and confirmed.

3) UAC detecting phase

As shown in Fig. 9, for 51,270 located dPairs, UACatcher detects 436 dPairs that lead to UAC bugs. We identify 346 of them as true positives and present the distribution detail in Table III. Among them, 337 are new bugs that were first disclosed by us. All of these bugs are under the reporting procedure and 277 have been confirmed and fixed by the community. With manual investigation into 90 false alarms, we find out that 33 of them come from inaccurate points-to sites. That is, the object being dereferenced is not necessarily the released object at runtime. For example, the deallocation sites of 26 cases not only release the object but also remove the object from a dynamic hashed list hence the dereference sites located by the points-to analysis actually access a different object in the same list. Additionally, other 57 cases are detected due to UACatcher's incomplete support of synchronizations and path constraints. Specifically, 29 of them are associated with reference counters that are difficult to analyze [41]. The remaining cases are associated with advanced synchronizations like semaphore, completion, and waitqueue. Due to their limited quantities, we decided to manually handle them.

Answer to RQ1: UACatcher is able to find and detect UAC bugs.

C. Estimate UAC Bugs (RQ2)

As mentioned in Section V, UACatcher regards whether a bug is exploitable according to two key points: (1) if the identified race window is large enough or could be manipulated; (2) if this UAC bug can be exploited via pseudo-terminal devices. Based on that, UACatcher finds 13 exploitable bugs among all detected true UAC bugs. as shown in Fig. 10. The listed table also gives the description and impact of these 13 bugs. Seven of them are found in the Bluetooth stack layer, two of them are from the NFC stack layer and the remaining four are from the AX25 net layer and MCTP layer. Importantly, except for the two bugs found in the NFC stack layer that require no privilege, the other nine bugs only require `CAP_NET_ADMIN` privilege. That is, none of them requires root privilege to exploit. We note that the ⋆ marked bugs in Fig. 10 own great exploitability. The dereference sites for these two bugs are not for instantly read or write action like the other bugs. These sites are used to obtain a child object that will perform a series of tasks. Therefore, with the heap spray approach, we can craft a malicious object and let our own object perform those tasks. If there is a code pointer in this child object, we can spray this pointer with the gadget address, hence a PC hijacking primitive is easily gained. As a matter of fact, we have developed a complete exploit for the sixth bug which gains a kernel space arbitrary code execution primitive and achieves the Local Privilege Escalation (LPE). In detail, this exploit makes use of the `userfaultfd` approach two times. The first registered page-fault handler acts as the routine switch point to hang the syscall routine and start the device cleanup routine. At the same time, another registered page-fault handler combines the `setxattr` approach, which is a popular system call for heap spraying [31], to fill the malicious payload into the deallocated object. Moreover, a thread is created and keeps emulating a new device for creating legal workqueues, because we expect the already destroyed workqueue to be fetched and used by the kernel. Otherwise, the exploitation fails as the hijacked code pointer registered in a destroyed workqueue is not going to be scheduled. This LPE exploit is included in the xairy's repo [42].

Answer to RQ2: UACatcher is able to estimate exploitable and highly risky UAC bugs.

D. Compare with DCUAF (RQ3)

UACatcher outperforms the existing tool [10][8] in two ways.

Bug Detections: We compare the capability of detecting UAC bugs between UACatcher and DCUAF [10] and the results are presented in Fig. 11. Specifically, the DCUAF only locates 16,832 dPairs and detects 9,820 from them. By manually investigating the result, we find that DCUAF doesn't find or detect any new results compared with UACatcher. That is, the located dPairs, the detected dPairs, as well as the 130 true bugs found by DCUAF are a true subset of UACatcher' results. The poor performance of DCUAF is mainly due to two reasons. Firstly, DCUAF and its solution mostly concentrate on the driver layer and should not be directly ported to other upper layers. Secondly, the lockset detection algorithm is insufficient for detecting UAC just as discussed in Section IV-C. We note that UACatcher's accuracy comes with a price: it takes about 8x time compared with the DCUAF because it concerns not only synchronizations like locks but also path constraints. We consider this price is affordable because UACatcher also identifies the race window of the bug which is extremely helpful for analyzing and estimating the bug.

Bug Estimation: As we discuss ahead, UACatcher is able to estimate the detected UAC bugs and find exploitable ones. In contrast, the state-of-the-art tool does not have this ability. It provides no additional information to help users to reproduce the detected bugs while UACatcher outputs routine switch details that help users to debug the race. In practice, UACatcher finds 13 exploitable bugs as shown in Fig. 10.

[8]Since this tool is not open source, we developed a QL version of it as a comparison

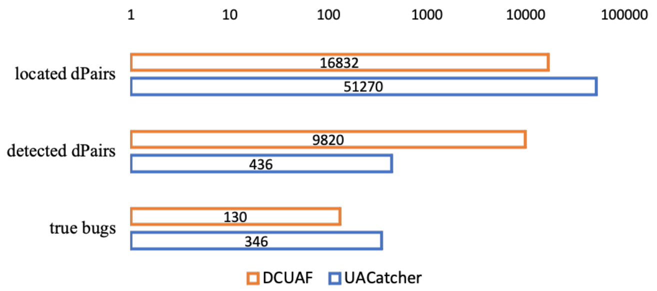


Fig. 11: Evaluation against the state-of-the-art.

To sum up, UACatcher outperforms the state-of-the-art tools since it can detect more UAC bugs with fewer false alarms and can also help estimate exploitable bugs.

Answer to RQ3: UACatcher outperforms the state-of-the-art tool DCUAF.

E. Adapt UACatcher to New Target (RQ4)

UACatcher is a portable tool that applies to different versions of the Linux kernel. It costs us no more than one hour to port UACatcher from 5.11 (git commit 7289e26f395b) kernel to 5.15 (git commit 583be982d934). Additionally, UACatcher successfully detects one **exploitable** UAC bug from the Management Component Transport Protocol (MCTP) layer that was first released in the 5.15 version (already shown in Fig. 10). This bug is quite similar to the bug we disclosed in the BT stack as this one can also be exploited with the `userfaultfd` approach.

Answer to RQ4: UACatcher can be ported to handle new targets easily.

VIII. Discussion

A. Analyzing the Kernel in Layers Granularity

For efficiency and accuracy sake, UACatcher conducts analysis and detection in layers granularity instead of the entire kernel. To accomplish that, UACatcher first finds driver layers based on pre-collected driver structures. It then finds upper layers based on call relations and heuristics-based characteristics. Even though the outcome is acceptable in practice, it brings out unnecessary false positives and false negatives. Additionally, we find layers with **irregular layer structure**. That is, unlike what is presented in Fig. 3, the found layer could have a more complex structure, e.g., mixed with multiple lower or upper layers. Since UACatcher currently does not support irregular layer structure, potential false negatives are raised. To avoid the above false results, we should give up the heuristics and spend more manual efforts on modeling the upper layers. We can dive into the kernel documentation and source code to gather the boundary functions of each subsystem into one database. Since this task is non-trivial, we leave it as future optimization.

B. Necessary Manual Efforts

Efforts for running UACatcher As mentioned in Section II-A and IV-A, UACatcher leverages the device driver model knowledge and cleanup routine knowledge to find driver layers. Within this procedure, we manually mark the pointer field responsible for device removal for all found driver structures. We note that this is accurate and low-cost due to the detailed comment of each driver type. Moreover, manual efforts are required to estimate exploitable UAC bugs. We have to manually prepare an emulated device and develop a Proof-of-Concept (PoC). To avoid duplication of work, we put the code for device emulation into one shared library and prepare PoC templates. There are tools [22,28,37] available that achieve automatic PoC or exploit generation. However, because most of these works rely on fuzzing techniques and cannot accurately handle the CUAF case, we prefer to write the PoC manually. Also, we have to manually prepare patches for all detected UAC bugs and work together with the community to fix them. Fortunately, since we summarized a set of rules to fix a UAC bug, this task is not difficult.

Efforts for evaluating UACatcher To achieve comprehensive evaluation, manual efforts are needed to investigate the results of each phase of UACatcher. As demonstrated in Section VII, since heuristics are used in the dPairs Preparing phase, we need to analyze the false negatives and false positives from the found layers and their boundary functions. Additionally, for all detected UAC bugs, we manually confirm the true positives. As pointed out in Section III, we believe manually analyzing, and even triggering these bugs is important. If we do not validate these, possible false alarms will cause an unnecessary burden to the kernel community. Specifically, the entire confirmation process costs roughly 80 human hours, which we regard as affordable.

C. *How to Fix and Prevent UAC bugs*

The direct solution to fix a UAC bug is to add appropriate constraint changes and checks, according to the examples shown in Fig. 6 and Fig. 7. The added code is supposed to satisfy the below requirements: (1) The constraint check site and the dereference site are protected by the same lock to make sure no context switch points can be injected between them. (2) The constraint change site should be placed before the deallocation site or after the deallocation site but in the same critical section. If these two requirements are satisfied, no routine switch points can be found to raise UAC bugs.

However, not all UAC bugs can be fixed with the aforementioned solution, as the lock used to protect the constraints may conflict with the existing locks. For example, we have confronted a bug that cannot be easily fixed since there is a `rwlock` (spinlock type) already held there while we need to add another `mutex_lock`. This is impossible since sleep-able locks like the `mutex_lock` are forbidden inside a `rwlock` to avoid blocking a CPU core. It took us and the maintainer weeks to work out a solution to fix this bug, which we called cleanup deferring. All the destructive actions that release resource objects are deferred to the final cleanup point in this fix[9]. This

[9]kernel commit e04480920d1 "Bluetooth: defer cleanup of resources in hci_unregister_dev"

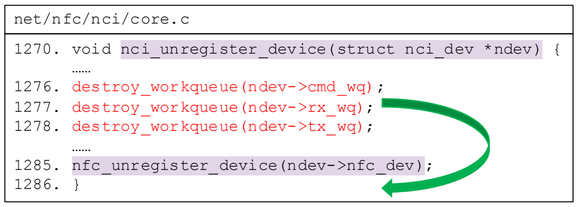

```
net/nfc/nci/core.c
void nci_unregister_device(struct nci_dev *ndev) {
......
destroy_workqueue(ndev->cmd_wq);
destroy_workqueue(ndev->rx_wq);
destroy_workqueue(ndev->tx_wq);
......
nfc_unregister_device(ndev->nfc_dev);
}
```

Fig. 12: The destructive actions that release the workqueue (marked in red) should be reordered to after the upper layer unreg-entry function `nfc_unregister_device`.

solution fixes the UAC bug by making sure that no resource is deallocated before a dereference action can occur.

Furthermore, our study finds that the destructive actions in the current layer are suggested to be performed after its upper layer has finished the cleanup routine. For example, Fig. 12 shows how the NCI layer should fulfill this requirement. The deallocation sites are originally placed before the unreg-entry function `nfc_unregister_device` of the upper NFC layer. This means that the resource can be destroyed even before the upper layer becomes aware of the device detachment. By deferring these three `destroy_workqueue` functions, no resources will be released until the upper layer finishes the cleanup.

From all the above solutions to fix an existing UAC bug, we can conclude the notes to prevent the introduction of new UAC bugs. Firstly, it is suggested to manage resource objects that are shared between different layers with a reference counter. This can efficiently hinder use-after-bug since the object will not be released once there is still an active reference. Additionally, for objects that are hard to incorporate with the reference counter, the deallocation sites should be deferred until the other layers are noticed with the device detachment or be protected with appropriate locks and constraints.

D. *Detecting Other CUAF Bugs*

Use-After-Cleanup (UAC) is defined as a CUAF bug that is triggered when a device is detached. As demonstrated in Fig. 1 and Fig. 3, the root cause of a UAC bug is the flaw in synchronization between the bottom-up cleanup routine and top-down syscall routine. For other CUAF bugs, the deallocation and dereference can be associated with other routines. That is, there can be different routine combinations such as routines both from devices (e.g., master device and slave device), and routines both from user space programs. (e.g., IPC and RPC). To extend UACatcher to detect CUAF bugs that are associated with other routines, the only required efforts are to find the new boundary functions for these routines. Once that task is fulfilled, UACatcher can locate the dPairs and detect true UAC bugs from them directly. Since it is non-trivial to model boundary functions out of the device domains, we leave this extension as future work.

IX. Related Work

A. *Detecting Concurrency Bugs in Kernel*

Static Analysis. Most static analysis-based tools use methods such as dataflow analysis, happens-before analysis, lockset analysis, and symbolic execution to detect concurrency bugs.

For example, RELAY [16] uses a relative lockset algorithm that could summarize functions that are independent of calling context to detect data races on millions of lines of code. RacerX [15] uses inter-procedural analysis to detect possible deadlocks and data races. DSAC [43] is a static analysis tool based on LLVM for detecting sleep-in-atomic-context concurrency bugs on kernel modules. It uses flow-sensitive and heuristics-based methods to detect bugs and generates fix commits automatically. DCUAF [10] is the state-of-the-art tool to detect CUAF bugs in Linux device drivers. It first adopts a local-global strategy to find concurrent interface pairs and then performs lockset analysis to detect CUAF bugs. However, the methods in these works fail to accurately detect UAC bugs.

Dynamic Analysis. The common dynamic analysis methods for detecting concurrency bugs include fuzzing, dynamic binary instrumentation, and so on. Redflag [44] is a kernel concurrency bugs detector based on dynamic binary instrumentation. It records useful information such as memory access operations caused by shared variables, and the use of synchronization functions. After that, it then uses lockset analysis and the algorithm of atomic violation to detect bugs. CONZZER [18] is a concurrency fuzzing framework that leverages context-sensitive and directional fuzzing approach for thread-interleaving exploration. With a customized mutation algorithm and breakpoint-control method, it can detect hard-to-find data races. Though many concurrency bugs are found by dynamic analysis methods, there are many limitations of using such methods to find UAC bugs. Firstly, most dynamic tools have no improvements to help explore states in the concurrency dimension hence rely on luck to trigger a UAC bug. More importantly, dynamic methods are generally not scalable, hence it will take too much effort to support all target layers found by UACatcher.

Hybrid Analysis. The hybrid analysis usually adopts both static analysis and dynamic analysis to strengthen the bug finding ability. KRACE [45] uses hybrid analysis to detect data race in the file system. It uses an evolutionary algorithm to generate inputs suitable for concurrency fuzzing and uses both alias coverage and branch coverage to guide fuzzing. Additionally, it collects information of coverage and execution logs and uses offline happen-before analysis and lockset analysis to detect the data races. RAZZER [46] is another hybrid race conditions detecting tool. It firstly uses static analysis to find the potential alias pairs in the kernel. Then, it uses fuzzing to generate system calls that could reach these alias pairs. Finally, it detects whether the alias pairs trigger a data race by hypervisor watchpoints. In future work, we can adopt similar techniques such as the log-based analysis and hypervisor watchpoints to help reproduce UAC bugs, which saves manual efforts.

B. Detecting and Mitigating Use-After-Free Bugs

Detecting Use-After-Free Bugs Most methods try to detect UAF bugs by dynamic analysis such as instrumentation or prediction. For example, UAFL [47] is a typestage-guided fuzzer. It first uses typestate analysis to extract the operation sequences that may lead to potential UAF vulnerabilities. It then instruments the target program to enable the collection of operation sequence coverage. This special coverage is used as feedback to guide the fuzzer. UFO [48] is a tool that detects UAF bugs in browsers. It first uses multi-threaded execution traces combined with happen-before relationships to speculate the maximal execution set. It further uses UAF constraints to find real UAF bugs from the execution set. Some methods use static analysis to detect UAF bugs. UAFchecker [49] adopts taint analysis and symbolic execution to make an inter-procedural analysis to find UAF bugs. Tac [50] uses SVM sorting algorithm to extract UAF-related aliases. It then uses pointer analysis to detect UAF bugs. Most of these methods only consider user space programs, hence it is difficult to port them to the kernel for detecting UAC bugs.

Mitigating Use-After-Free Bugs. Some solutions attempt to mitigate the UAF by taking special care of dangling pointers. For example, MarkUs [51] is a memory allocator that could prevent UAF bugs. It quarantines data free and prevents its reallocation until there is no dangling pointer pointing to it. Other solutions attempt to inject a check before dereferencing the pointers. Watchdog [52] generates a pointer identifier based on hardware for each memory that has been allocated and checks if the identifier is still valid before pointer dereference.

X. Conclusion

In this paper, we proposed UACatcher, the first tool that fills the gap in UAC bug detection. It first scans the entire kernel to find target layers and collects essential information for them. Next, it adopts the context- and flow-sensitive inter-procedural analysis and the points-to analysis to locate possible dPairs that can cause UAC bugs. It finally uses an algorithm named *routine switch point*, to detect the UAC bugs and estimate exploitable ones. In evaluation, UACatcher detects 346 true bugs. All the found bugs are reported to the Linux community and 277 of them have been confirmed and fixed and 15 CVEs have been assigned by the time of writing this paper (July 2022). Additionally, we find 13 exploitable UAC bugs that can be triggered via pseudoterminal-based device emulation technique. Therefore, they can be exploited and used to achieve the arbitrary code execution primitive in kernel space and achieve privilege escalation.

Acknowledgment

We would like to thank the anonymous reviewers and shepherd for their insightful comments that greatly helped improve the presentation and evaluation of the paper. We also thank the Linux kernel maintainers, especially Tetsuo Handa, Luiz Augusto von Dentz, and Krzysztof Kozlowski, who gave us feedback and help to fix the reported UAC bugs. This work was partially supported by the National Key R&D Program of China (No. 2022YFE0113200). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "Syzkaller, use-after-free read in blkcg_print_stat," https://groups.google.com/g/syzkaller-bugs/c/2YJt3Pwl92Y/m/p5TltZdsAgAJ.

[2] "Syzkaller, use-after-free read in ax88172a_unbind," https://groups.google.com/g/syzkaller-bugs/c/V7YReIFMOhk/m/UcmFU6cRAQAJ.

[3] D. Vyukov, "Syzkaller: an unsupervised, coverage-guided kernel fuzzer," 2019.

[4] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 167–182.

[5] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[6] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 729–743.

[7] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[8] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[9] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[10] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 255–268.

[11] "Ql language." https://securitylab.github.com/tools/codeql.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[13] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with portend," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 185–198, 2012.

[14] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How {Double-Fetch} situations turn into {Double-Fetch} vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1–16.

[15] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.

[16] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *ESEC-FSE '07*, 2007.

[17] "The kernel concurrency sanitizer (kcsan)." https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[18] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," 2022.

[19] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 166–177.

[20] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1007–1024.

[21] "Data-race detection in the linux kernel," https://man7.org/linux/man-pages/man7/pty.7.html, accessed: 2020-08-24.

[22] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.

[23] J. Ye, C. Zhang, and X. Han, "Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1529–1531.

[24] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[25] J. Huang, "Ufo: predictive concurrency use-after-free detection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 609–619.

[26] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1867–1881. [Online]. Available: https://doi.org/10.1145/3319535.3354244

[27] T. Li, J.-J. Bai, Y. Sui, and S.-M. Hu, "Path-sensitive and alias-aware typestate analysis for detecting os bugs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 859–872. [Online]. Available: https://doi.org/10.1145/3503222.3507770

[28] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 781–797.

[29] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 446–455, 2007.

[30] jannh, "Linux: Uaf via double-fdput() in bpf(bpf_prog_load) error path." https://bugs.chromium.org/p/project-zero/issues/detail?id=808.

[31] "Linux kernel universal heap spray," https://duasynt.com/blog/linux-kernel-heap-spray, 2018.

[32] "Linux. ptmx(4) - linux man page." https://linux.die.net/man/4/ptmx.

[33] "pseudoterminal interfaces linux manual page," https://man7.org/linux/man-pages/man7/pty.7.html, accessed: 2020-08-13.

[34] "Pseudoterminal," https://lpc.events/event/7/contributions/647/attachments/549/972/LPC2020-KCSAN.pdf, last edited: 2021-11-13.

[35] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[36] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 19–36.

[37] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1187–1204.

[38] "Codeql for research." https://securitylab.github.com/tools/codeql.

[39] "Codeql: the libraries and queries," https://github.com/github/codeql, 2021.

[40] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[41] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with {Two-Dimensional} consistency checking," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2471–2488.

[42] "linux kernel exploitation." https://github.com/xairy/linux-kernel-exploitation.

[43] J.-J. Bai, Y. Wang, J. L. Lawall, and S. Hu, "Dsac: Effective static analysis of sleep-in-atomic-context bugs in kernel modules," in *USENIX Annual Technical Conference*, 2018.

[44] J. Seyster, P. Radhakrishnan, S. Katoch, A. Duggal, S. D. Stoller, and E. Zadok, "Redflag: A framework for analysis of kernel-level concurrency," in *ICA3PP*, 2011.

[45] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1643–1660, 2020.

[46] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768, 2019.

[47] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 999–1010, 2020.

[48] J. Huang, "Ufo: Predictive concurrency use-after-free detection," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 609–619, 2018.

[49] J. Ye, C. Zhang, and X. Han, "Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities," *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[50] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017.

[51] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 578–591, 2020.

[52] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 189–200, 2012.

APPENDIX

Table I: Covered Driver Types and Locations

name	**file**
acpi_driver	include/acpi/acpi_bus.h
pci_driver	include/linux/pci.h
platform_driver	include/linux/platform_device.h
eisa_driver	include/linux/eisa.h
ssb_driver	include/linux/ssb/ssb.h
bcma_driver	include/linux/bcma/bcma.h
pnp_driver	include/linux/pnp.h
pnp_card_driver	include/linux/pnp.h
i2c_driver	include/linux/i2c.h
spi_driver	include/linux/spi/spi.h
virtio_driver	include/linux/virtio.h
scsi_driver	include/scsi/scsi_driver.h
nd_device_driver	include/linux/nd.h
pcmcia_driver	include/pcmcia/ds.h
parport_driver	include/linux/parport.h
auxiliary_driver	include/linux/auxiliary_bus.h
isa_driver	include/linux/isa.h
phy_driver	include/linux/phy.h
i3c_driver	include/linux/i3c/device.h
sdw_driver	include/linux/soundwire/sdw.h
slim_driver	include/linux/slimbus.h
spmi_driver	include/linux/spmi.h
xenbus_driver	include/xen/xenbus.h
usb_driver	include/linux/usb.h
sdio_driver	include/linux/mmc/sdio_func.h
serdev_device_driver	include/linux/serdev.h
fsl_mc_driver	include/linux/fsl/mc.h
mhi_driver	include/linux/mhi.h
moxtet_driver	include/linux/moxtet.h
scmi_driver	include/linux/scmi_protocol.h
dax_device_driver	drivers/dax/bus.h
pci_epf_driver	include/linux/pci-epf.h

Continued on next page

Table I: Covered Driver Types and Locations (Continued)

idxd_device_driver	drivers/dma/idxd/idxd.h
fw_driver	include/linux/firewire.h
tee_client_driver	include/linux/tee_drv.h
coreboot_driver	drivers/firmware/google/coreboot_table.h
dfl_driver	drivers/fpga/dfl.h
fsi_driver	include/linux/fsi.h
mcb_driver	include/linux/mcb.h
siox_driver	include/linux/siox.h
mipi_dsi_driver	include/drm/drm_mipi_dsi.h
drm_i2c_encoder_driver	include/drm/drm_encoder_slave.h
mdev_driver	include/linux/mdev.h
amba_driver	include/linux/amba/bus.h
greybus_driver	include/linux/greybus.h
hid_driver	include/linux/hid.h
hv_driver	include/linux/hyperv.h
rmi_driver	include/linux/rmi.h
usb_device_driver	include/linux/usb.h
ishtp_cl_driver	include/linux/intel-ish-client-if.h
hsi_client_driver	include/linux/hsi/hsi.h
intel_th_driver	drivers/hwtracing/intel_th/intel_th.h
serio_driver	include/linux/serio.h
ide_driver	include/linux/ide.h
gameport_driver	include/linux/gameport.h
ipack_driver	include/linux/ipack.h
bttv_sub_driver	drivers/media/pci/bt8xx/bttv.h
rpmsg_driver	include/linux/rpmsg.h
radio_isa_driver	drivers/media/radio/radio-isa.h
memstick_driver	include/linux/memstick.h
tifm_driver	include/linux/tifm.h
mei_cl_driver	include/linux/mei_cl_bus.h
mmc_driver	drivers/mmc/core/bus.h
spi_mem_driver	include/linux/spi/spi-mem.h
mdio_driver	include/linux/mdio.h
tc_driver	include/linux/tc.h
tb_service_driver	include/linux/thunderbolt.h
pcie_port_service_driver	drivers/pci/pcie/portdrv.h
ulpi_driver	include/linux/ulpi/driver.h
asus_wmi_driver	drivers/platform/x86/asus-wmi.h
wmi_driver	include/linux/wmi.h
rio_driver	include/linux/rio.h
apr_driver	include/linux/soc/qcom/apr.h
usb_gadget_driver	include/linux/usb/gadget.h
tty_ldisc_ops	include/linux/tty_ldisc.h
anybuss_client_driver	drivers/staging/fieldbus/anybuss/anybuss-client.h
gbphy_driver	drivers/staging/greybus/gbphy.h
visor_driver	include/linux/visorbus.h

Continued on next page

Table I: Covered Driver Types and Locations (Continued)

bcm2835_audio_driver	drivers/staging/vc04_services/bcm2835-audio/bcm2835.c
vme_driver	include/linux/vme.h
usb_composite_driver	include/linux/usb/composite.h
snd_seq_driver	include/sound/seq_device.h
usb_serial_driver	include/linux/usb/serial.h
typec_altmode_driver	include/linux/usb/typec_altmode.h
vdpa_driver	include/linux/vdpa.h
hdac_driver	include/sound/hdaudio.h
hda_codec_driver	include/sound/hda_codec.h
ac97_codec_driver	include/sound/ac97/codec.h

Table II: Collected Deallocation Functions.

name	**file**
destroy_workqueue	include/linux/workqueue.h
kfree_sensitive	include/linux/slab.h
kfree	include/linux/slab.h
kmem_cache_free	include/linux/slab.h
kvfree	include/linux/mm.h
kfree_const	include/linux/string.h
kfree_skb	include/linux/skbuff.h
kfree_sensitive	mm/slab_common.c
kfree	mm/slub.c
crypto_free_shash	include/crypto/hash.h
kvfree	mm/util.c
kmem_cache_free	mm/slub.c
kfree_skb	net/core/skbuff.c
kfree_const	mm/util.c
destroy_workqueue	kernel/workqueue.c
rfkill_destroy	net/rfkill/core.c

TABLE III: Distribution of the found true UAC Bugs: Among them, 346 are found in the 5.11 kernel and an additional one (given in the last line) is found in the 5.15 kernel. Refer to Sec II for the detail meaning of the field **Layer Type**.

Domain	**Layer Type**	**Code Directory**	**UAC Bugs**
ATM device	driver (pci_driver)	drivers/atm/	22
Conexant cx23418 multimedia device	driver (pci_driver)	drivers/media/pci/cx18	1
Chelsio network device	driver (pci_driver)	drivers/net/ethernet/chelsio/cxgb4/	3
Intel network device	driver (pci_driver)	drivers/net/ethernet/intel/igb/	30
QLogic network device	driver (pci_driver)	drivers/net/ethernet/qlogic/qede/	1
Skylake/SST sound device	driver (pci_driver)	sound/soc/intel/skylake/	17
VSP1 video device	driver (platform_driver)	drivers/media/platform/vsp1/	4
WIZnet devices	driver (platform_driver)	drivers/net/ethernet/wiznet/	1
cJTAG misc device	driver (pti_pci_remove)	drivers/misc/	7
Keystream device	driver (sdio_driver)	drivers/staging/ks7010/	7
IEEE802154 device	driver (spi_driver)	drivers/net/ieee802154/	1
Hamradio device	driver (tty_ldisc_ops)	drivers/net/hamradio/	23
AX25 netstack	upper	net/ax25/	118
BT netstack	upper	net/bluetooth/	88
DVB core	upper	drivers/media/dvb-core/	1
MAC802154 netstack	upper	net/mac802154/	1
NFC NCI stack	upper	net/nfc/nci/	7
NCSI netstack	upper	net/ncsi/	6
NFC netstack	upper	net/nfc/	1
STM core	upper	drivers/hwtracing/stm/	7
MCTP netstack	*upper*	*net/mctp*	*1*