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

Kernel Vulnerability Mining

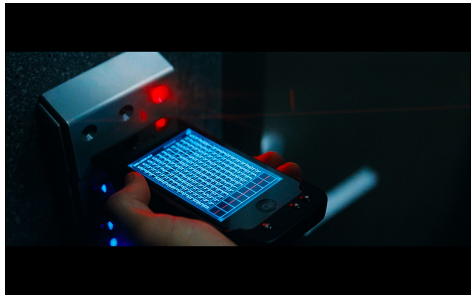
- **From user-space side (Syscall)**
  - syzkaller[1], Difuze[CCS '17], Healer[SOSP '21] …
- From device side (Interrupt, DMA)
  - syzkaller-external (usb/bt/nfc/wireless[2]), PrintFuzz[ISSTA '22] …

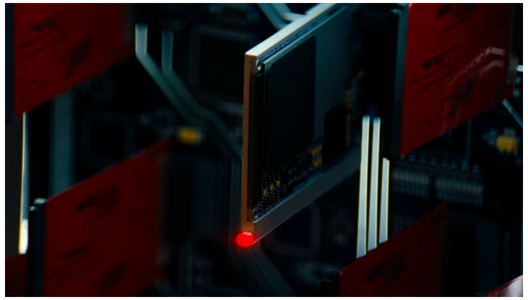
[1] https://github.com/google/syzkaller

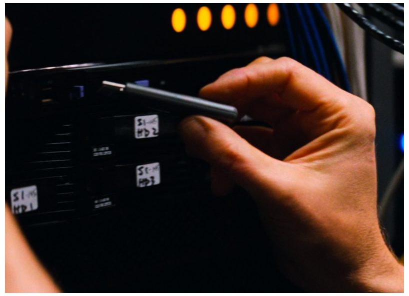
[2] https://github.com/google/syzkaller/blob/master/docs/linux/wifi_fuzzing.md

Kernel Vulnerability Mining

- From user-space side (Syscall)
  - syzkaller[1], Difuze[CCS '17], Healer[SOSP '21] …
- **From device side (Interrupt, DMA)**
  - syzkaller-external (usb/bt/nfc/wireless[2]), PrintFuzz[ISSTA '22] …







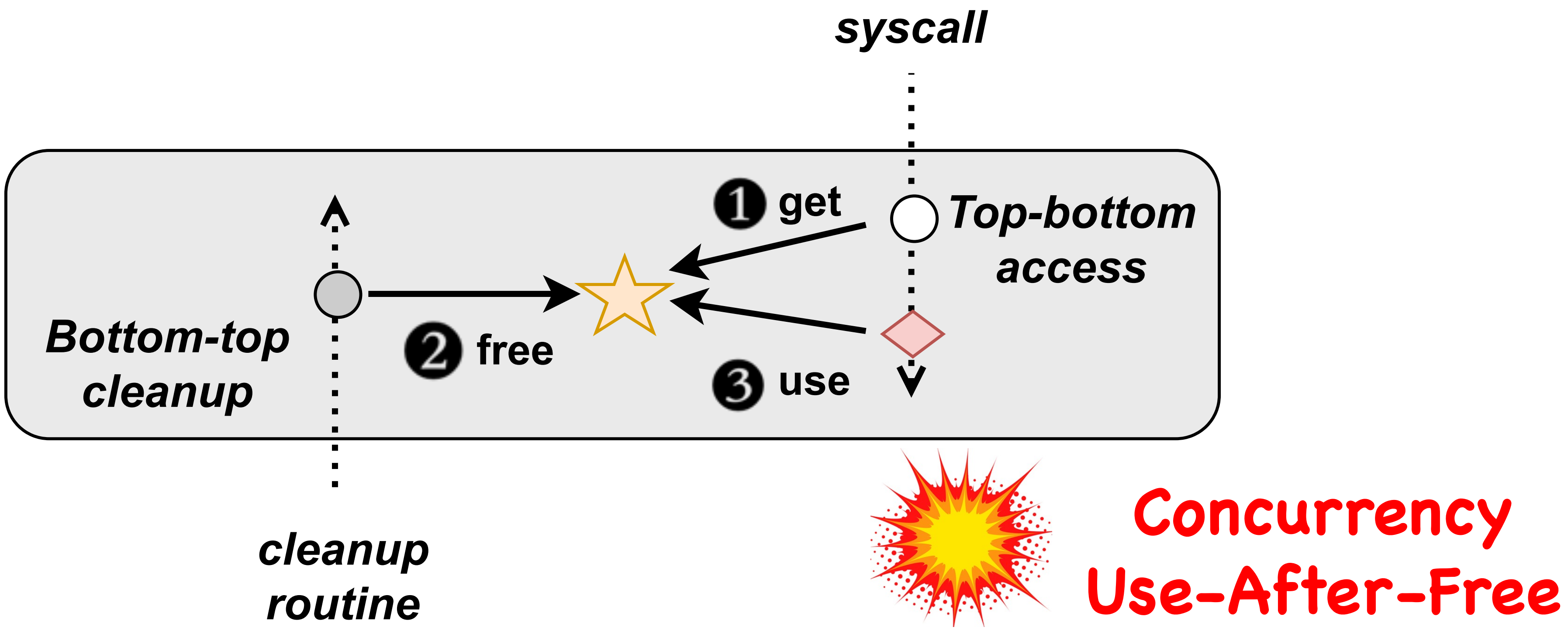
Kernel Vulnerability Mining

- **From user-space side (Syscall)**
  - syzkaller[1], Difuze[CCS '17], Healer[SOSP '21] …
- **From device side (Interrupt, DMA)**
  - syzkaller-external (usb/bt/nfc/wireless[2]), PrintFuzz[ISSTA '22] …

somehow combine?

UAC (Use-After-Cleanup)

When a device is **detached**, the running syscall may be unaware and **attempt to use** an already **released** object



However

The community does not pay much attention to UAC bug, which **was improperly treated as a low-security impact**

- Concurrency issue seems need some luck to trigger 🤷‍♂️
- Detachment seems need real hardware ⚙️
  (or root-privilege-required virtual device)

However again, CVE-2021-3573

- ~~Concurrency issue seems need luck to trigger~~

  Stably triggered 👍
- ~~Detachment seems need real hardware~~

  From low-privilege user-space 👍

and Achieve Local Privilege Escalation (LPE) Attack

check demo[1] here and details[2] here

Blue Klostski

[1] https://github.com/uacatcher/uacatcher-repo/blob/main/demo.gif

[2] https://f0rm2l1n.github.io/2021-07-23-Blue-Klotski/

Motivations

1. UAC bug is exploitable and with high risks

   which is different from the widely thought

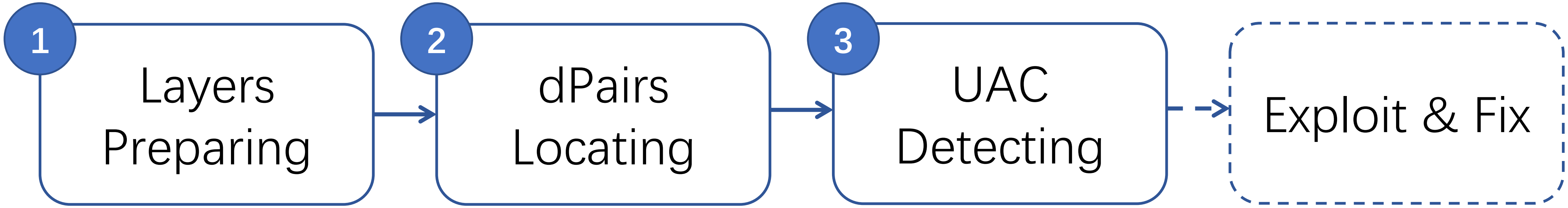
2. UAC bug cannot be systematically detected by existing tools

- Not for race at all (deadline[SP '18], DASC[ATC '18], …)
- Just for data race (Razzer[SP '19], Krace[SP '20], …)
- For CUAF but not concern cleanup (DCUAF[ATC '19])

Our Solution: UACatcher

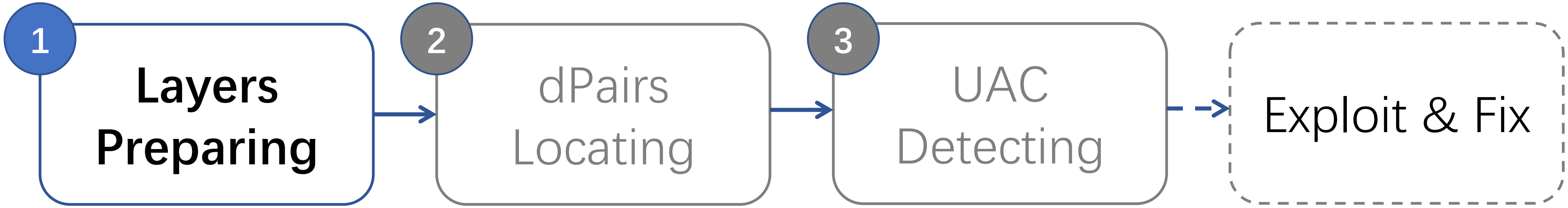
- Detect Use-After-Cleanup Bug via **Static Analysis**
- and Estimate **Exploitable** Bugs from detected ones

Three Main Phases Overview

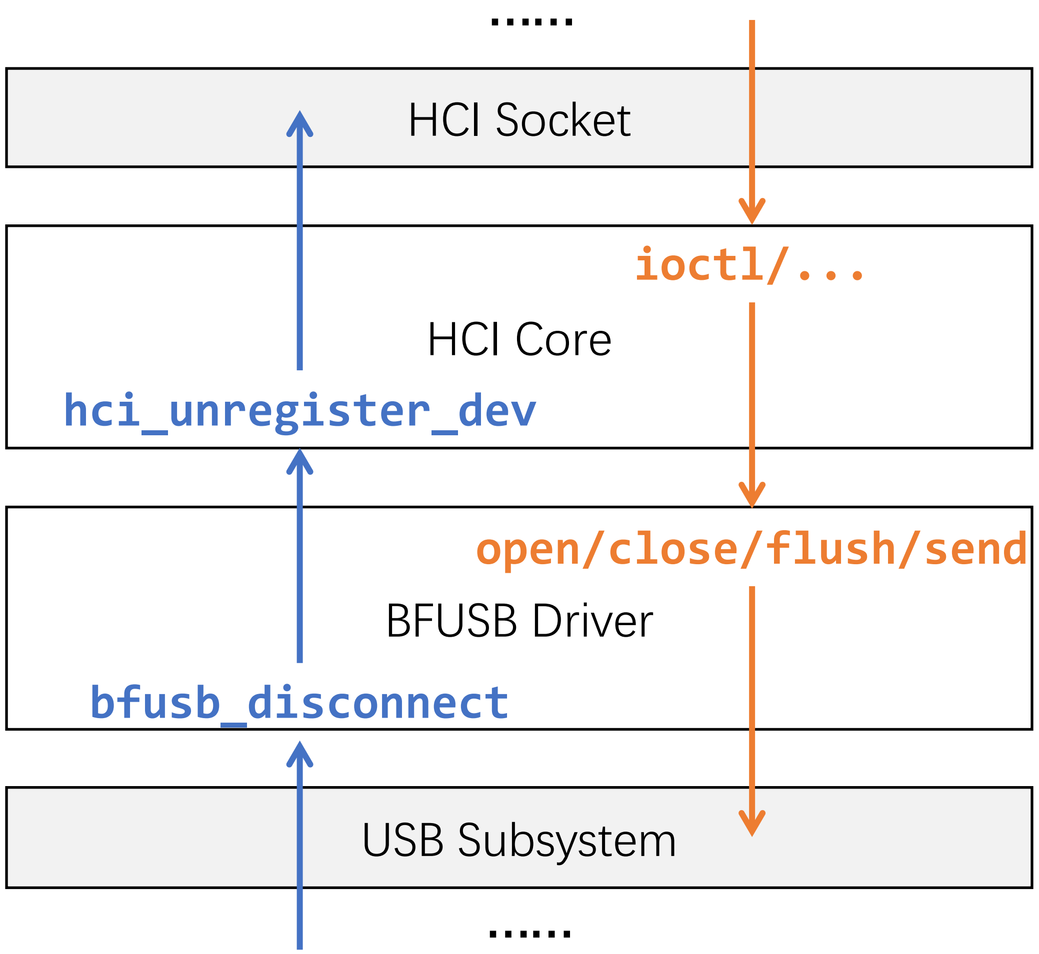


1 Layers Preparing

- Analyzing the entire kernel is too **expensive** for static analysis
  - the kernel follows a **layered architecture**
  - routines can be **split** into **layers granularity**



Example: BT



- unreg-entry functions
  - cleanup routine
- interface functions
  - syscall routine

- Collectively referred to as **boundary functions**
- Found via **model** and **heuristics** methods

Example: BT - Find driver unreg-entry functions

```
struct device_driver {
  ...
  int (*remove) (struct device* dev)
  ...
};

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

struct usb_driver {
  ...
  void (*disconnect) (...)
  ...
  struct usbdrv_wrap drvwrap;
  ...
}
```

1. find the driver type `(usb_driver)`
2. annotate the pointer field for handling cleanup `(disconnect)`
3. derive the actual unreg-entry function from specific driver struct `(bfusb_disconnect from bfusb_driver)`

```
static struct usb_driver bfusb_driver =
{
   .disconnect    = bfusb_disconnect,
   ...
}
```

Example: other BT boundary functions

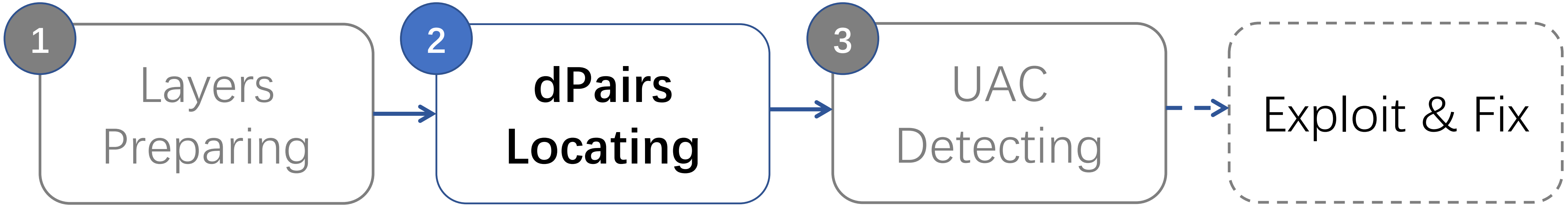
With the driver layer unreg-entry pinpointed

- Find the above layers unreg-entry from bottom-up via **characteristics-based heuristics**
  - `void hci_unregister_dev(struct hci_dev *hdev)`
- Find the interface functions from top-down as **packed code pointers [1]**
  - `open/close/flush/send` from `struct hci_dev`

[1] Zhang, Tong, et al. "Pex: A permission check analysis framework for linux kernel." *28th USENIX Security Symposium*. 2019.

2 dPairs Locating

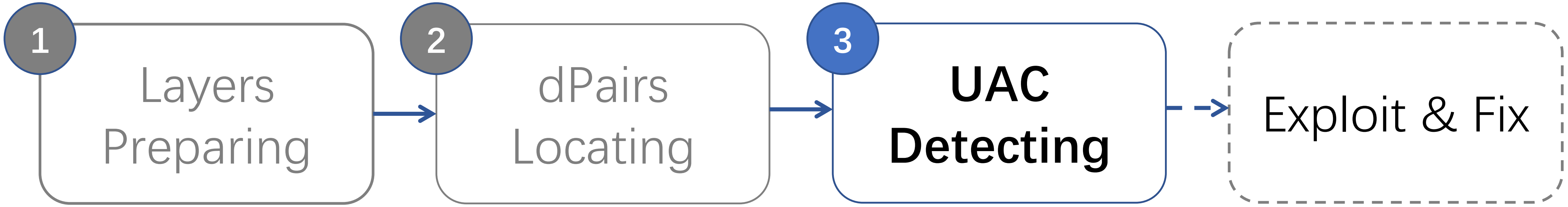
- Locate all **deallocation** & **dereference** site pairs (dPairs)
  - Track all call sites to dealloc functions (e.g., kfree)
  - Leverage points-to analysis
  - *Add filters to discard false result*



3 UAC Detecting

- Detect if a located dPair causes UAC or not
  - *routine-switch-point algorithm*

- Also **Estimate** detected bugs



Example: UAC-free code

cleanup routine

```
1 mutex_lock(&ndev->req_lock)
2 if !test_and_clear_bit(NCI_UP, &ndev->flags)
      return
...
3 mutex_unlock(&ndev->req_lock)
4 destroy_workqueue(ndev->cmd_wq)
  // deallocation
...
```

syscall routine

```
5 mutex_lock(&ndev->req_lock)
6 if !test_bit(NCI_UP, &ndev->flags)
      goto 8
...
7 queue_work(ndev->cmd_wq, ...)
  // dereference
...
8 mutex_unlock(&ndev->req_lock)
```

Use **synchronizations** and **path constraints** to make sure

- *dereference would not happen after the deallocation*

Routine-switch-point Algorithm

Find the routine-switch point that

- schedule dereference after the deallocation

Forward & Backward to get all path constraints ***P***

- If ***P*** is satisfy-able, the dPair cause true UAC
- otherwise, safe one

check more details in the paper ☺

UAC Bug Estimation

1. Race Window **Identification** and **Measurement**

   concern on **time-consuming functions** `(memory allocation/IO/...)` and **time-controllable functions** `(copy_{from/to}_user)`

2. User-space Device Emulation

   by **Pseudoterminal Device** and **Line Discipline**

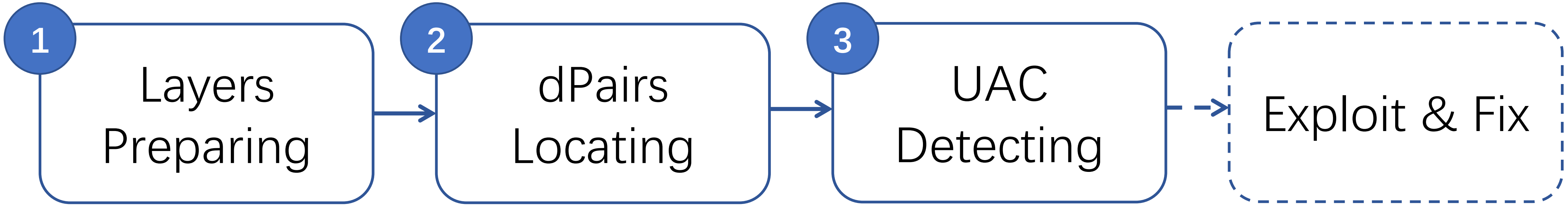
   support BT/NFC/HAMRADIO/CAN devices for now

Implementation & Evaluation Setup

- CodeQL Queries
  - port from user-space to Linux kernel
- Python Code
  - driver for running QL and fetching result
  - CallGraph, CFG
  - routine-switch point algoirhm
- Evaluated with Linux kernel 5.11 (7289e26f395b)

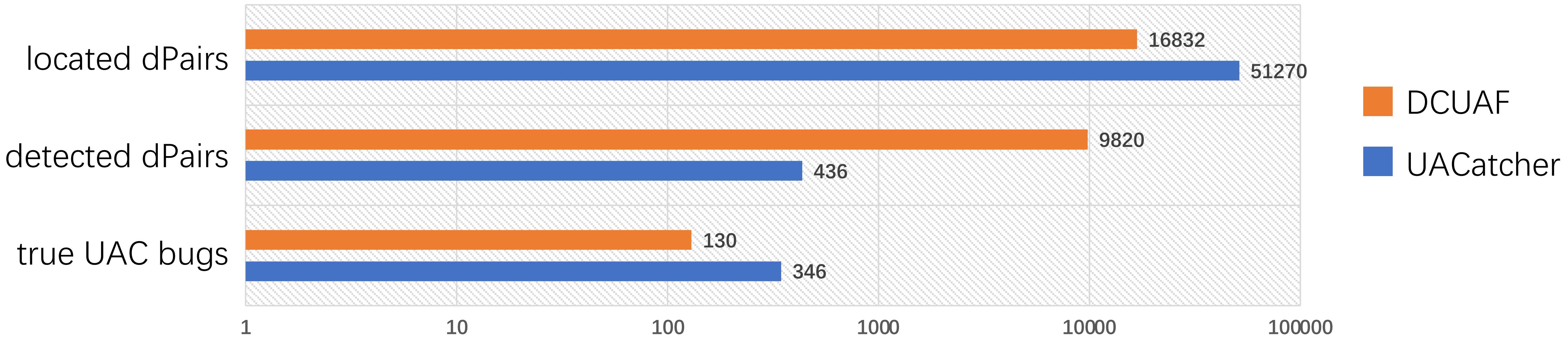
Evaluation Result (for each phase)

- **Layers Preparing**
  - **88** driver types, **1,856** layers
- **dPairs Locating Phase**
  - **136,628** dPairs located
  - **51,270** after filterring
- **UAC Detecting Phase**
  - **436** detected UAC
  - **346** true bugs
  - **277** been confirmed
  - **15** CVEs assigned



Evaluation Result (compare with DCUAF[1])

- **Locate more dPairs**: 51,270 > 16,832
- **Detect more real UAC bugs**: 346 > 130
- **Get less False Positives**: 20.6% < 98.6%



[1]*Not open source, we developed a QL version of it as a comparison*

Evaluation Result (estimation)

13 exploitable UAC bugs (one from **5.15** kernel)

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	□

□ read from a destroyed object
◊ write to a destroyed object
△ cause Denial of Service
⋆ complex task

Conclusions

- UAC bugs are **high risks**
- UACatcher is the **first** tool that **systematically** detects UAC bugs in Linux kernel
- **346** true bugs found, **277** fixed, **15** CVEs assigned
- **13** exploitable UAC estimated

Thank you!

Lin MA, ZheJiang University
linma@zju.edu.cn