HFL: Hybrid Fuzzing on the Linux Kernel

Kyungtae Kim*, Dae R. Jeong°, Chung Hwan Kim¶,
Yeongjin Jang§, Insik Shin°, Byoungyoung Lee˥*

**Purdue University, °KAIST, ¶NEC Labs America,*
§Oregon State University, ˥Seoul National University

Software Security Analysis

- Random fuzzing
  - **Pros**: Fast path exploration
  - **Cons**: Strong branch conditions e.g., *if(i == 0xdeadbeef)*
- Symbolic/concolic execution
  - **Pros**: Generate concrete input for strong branch conditions
  - **Cons**: State explosion

Hybrid Fuzzing in General

- Combining ***traditional fuzzing*** and ***concolic execution***
  - *Fast exploration* with fuzzing (*no state explosion*)
  - *Strong branches are handled* with concolic execution

- State-of-the-arts
  - Intriguer [CCS'19], DigFuzz [NDSS'19], QSYM [Sec'18], etc.
  - Application-level hybrid fuzzers

Kernel Testing with Hybrid Fuzzing

- Software vulnerabilities are critical threats to OS kernels
  - ***1,018 Linux kernel vulnerabilities*** *reported in CVE over the last 4 years*

- Hybrid-fuzzing can help improve coverage and find more bugs in kernels.
  - A huge number of specific branches e.g., CAB-Fuzz[ATC’17], DIFUZE[CCS’17]

Kernel Testing with Hybrid Fuzzing

- Software vulnerabilities are critical threats to OS

Q. Is hybrid-fuzzing good enough for kernel testing?

more bugs in kernels.
  - A huge number of specific branches e.g., CAB-Fuzz[ATC’17], DIFUZE[CCS’17]

Challenge 1: Indirect Control Transfer

derived from
syscall arguments

```
idx = cmd - INFO_FIRST;
    ...
funp = _ioctls[idx];
    …
funp (sbi, param);
```

<indirect function call>

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
          ...
    ioctl_ismountpoint,
};
```

<function pointer table>

Challenge 1: Indirect Control Transfer

derived from
syscall arguments

```
idx = cmd - INFO_FIRST;
    ...
funp = _ioctls[idx];
    ...
funp (sbi, param);
```

<indirect function call>

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
        ...
    ioctl_ismountpoint,
};
```

<function pointer table>

Challenge 1: Indirect Control Transfer

derived from
syscall arguments

```
idx = cmd - INFO_FIRST;
    ...
funp = _ioctls[idx];
    ...
funp (sbi, param);
```

<indirect function call>

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
        ...
    ioctl_ismountpoint,
};
```

<function pointer table>

Challenge 1: Indirect Control Transfer

derived from
syscall arguments

```
idx = cmd - INFO_FIRST;
    ...
funp = _ioctls[idx];
    ...
funp (sbi, param);
```

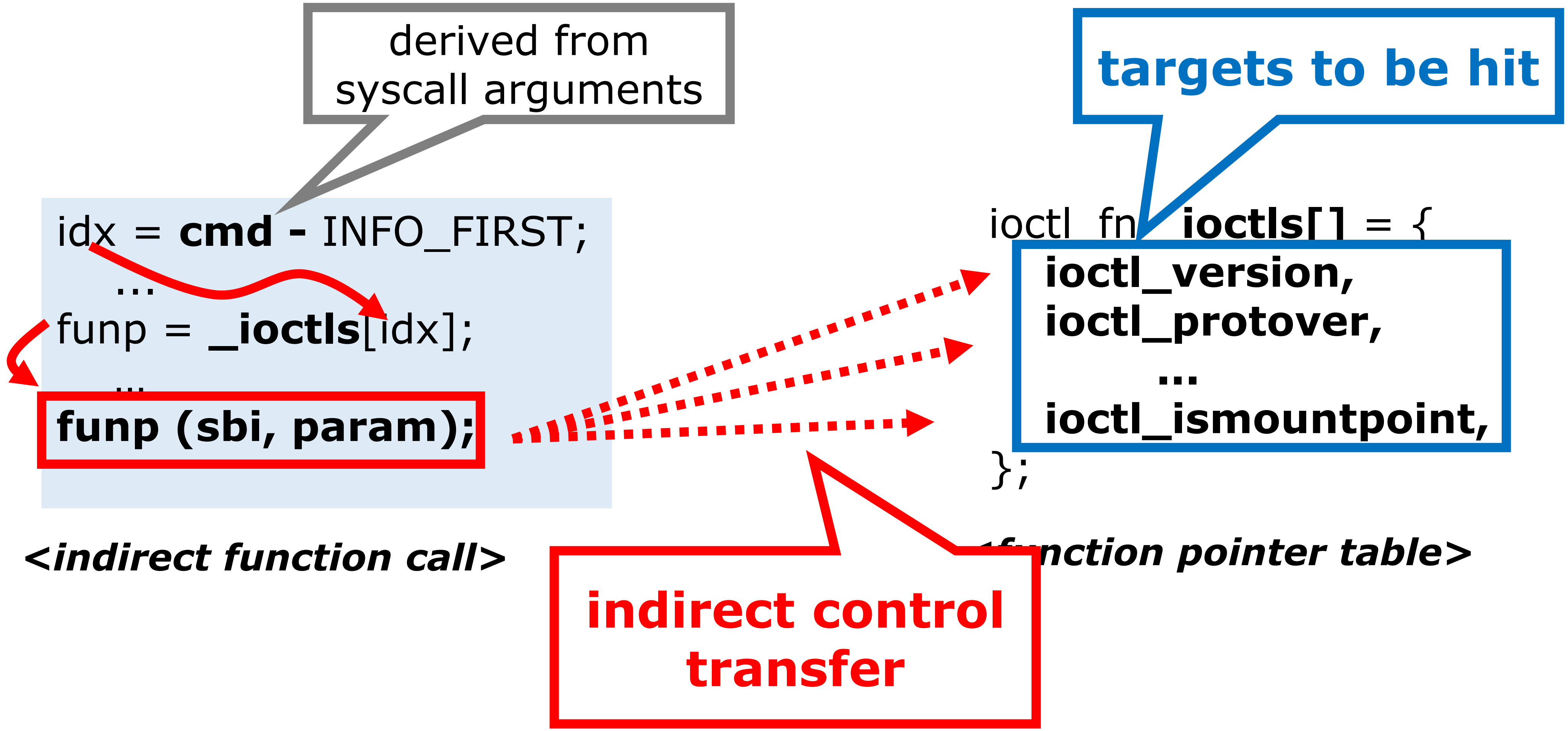
<indirect function call>

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
        ...
    ioctl_ismountpoint,
};
```

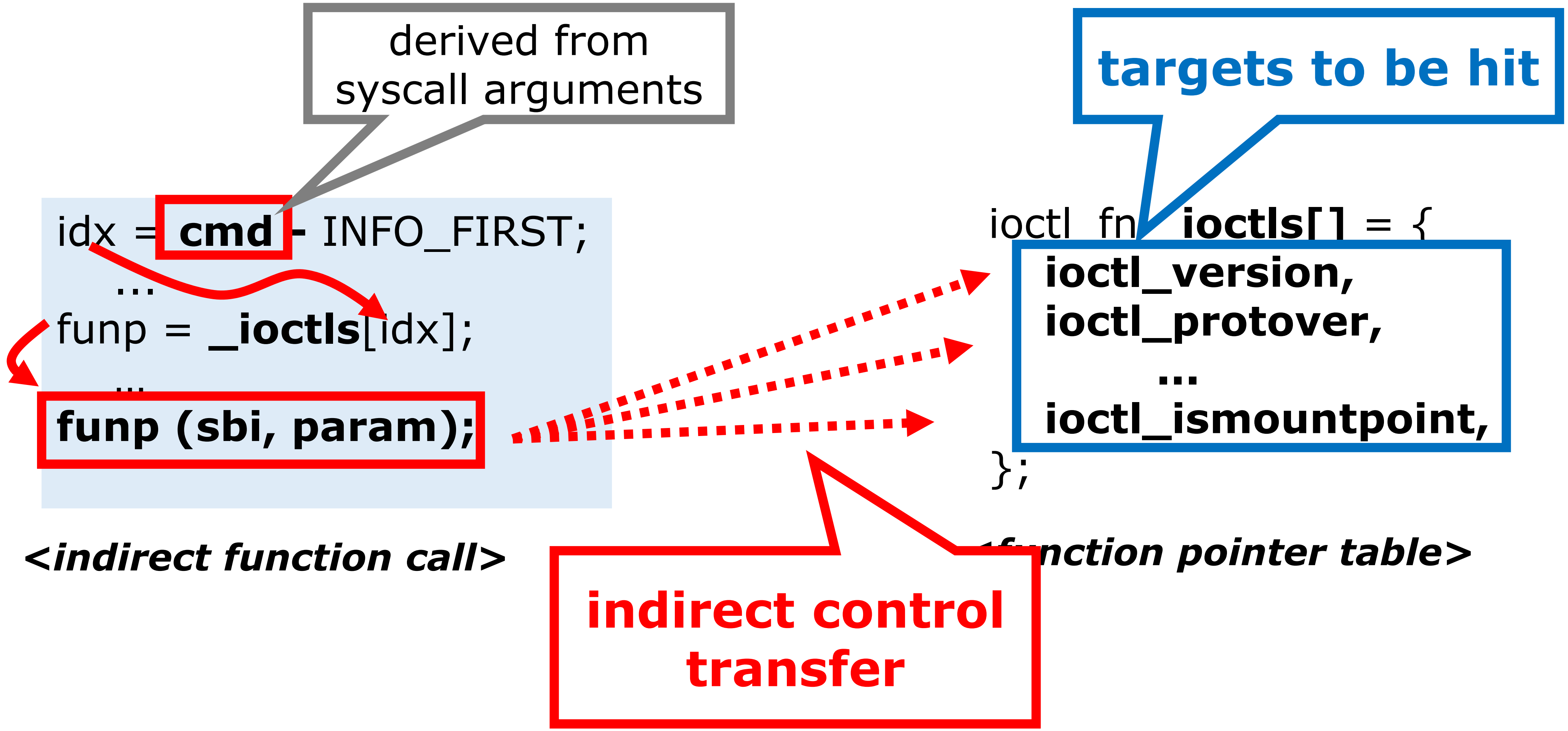
<function pointer table>

indirect control transfer

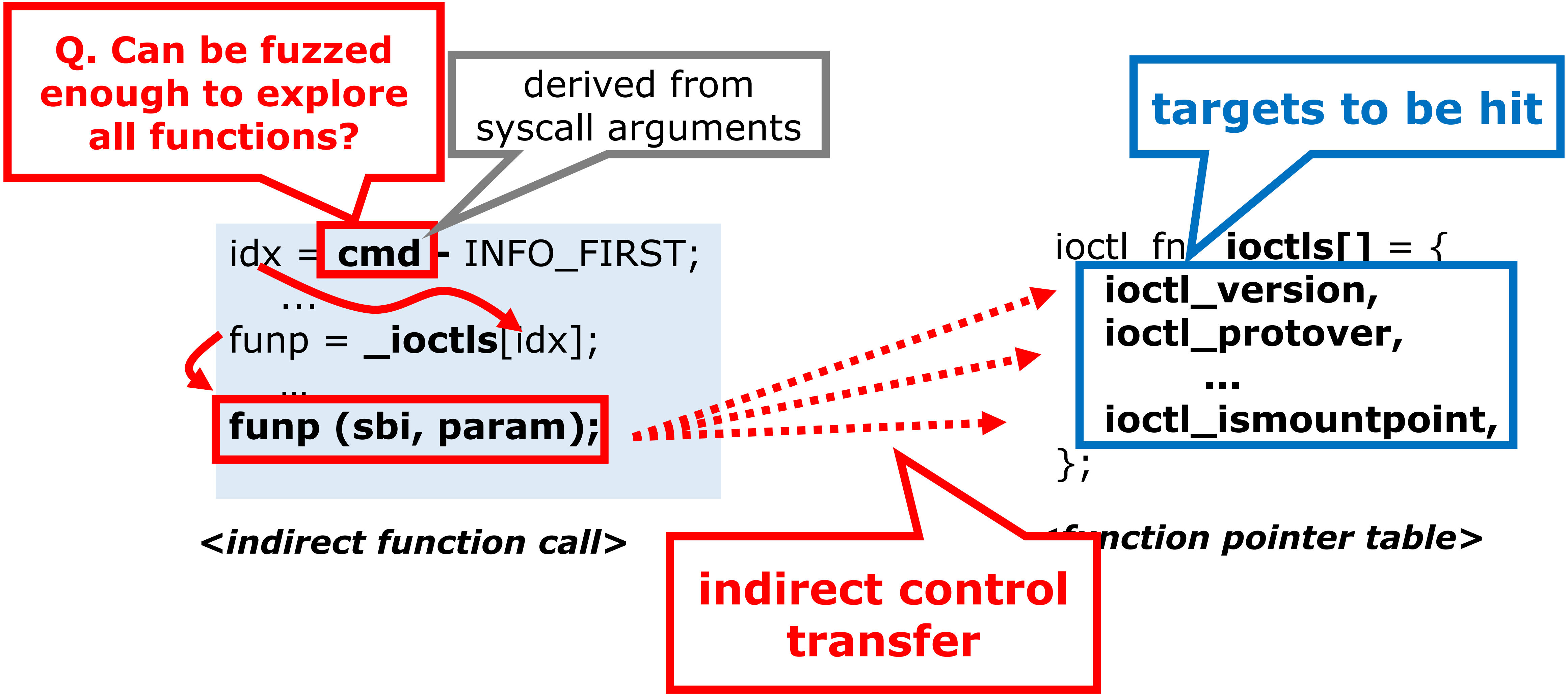
Challenge 1: Indirect Control Transfer



Challenge 1: Indirect Control Transfer



Challenge 1: Indirect Control Transfer



Challenge 2: System Call Dependencies

__int__ ***open*** *(const char *pathname, int flags, mode_t mode)*
ssize_t ***write*** *(__**int fd**__, void *buf, size_t count)*

ioctl *(int fd, unsigned long req, void *argp)*
ioctl *(int fd, unsigned long req, void *argp)*

Challenge 2: System Call Dependencies

explicit syscall dependencies

***int* open** *(const char *pathname, int flags, mode_t mode)*
ssize_t ***write*** *(**int fd**, void *buf, size_t count)*

ioctl *(int fd, unsigned long req, void *argp)*
ioctl *(int fd, unsigned long req, void *argp)*

Challenge 2: System Call Dependencies

explicit syscall dependencies

***int* open** *(const char *pathname, int flags, mode_t mode)*
ssize_t ***write*** *(**int fd**, void *buf, size_t count)*

ioctl *(int fd, unsigned long req, void *argp)*
ioctl *(int fd, unsigned long req, void *argp)*

Q. What dependency behind?

Example: System Call Dependencies

```
fd = open (…)
ioctl (fd, D_ALLOC, arg1)
ioctl (fd, D_BIND, arg2)
```

Example: System Call Dependencies

fd = open (…)
ioctl (fd, D_ALLOC, ***arg1****)*
ioctl (fd, D_BIND, ***arg2****)*

1 first *ioctl*

ioctl (fd, cmd, arg):
switch (cmd) {
case D_ALLOC: **d_alloc** (arg);
case D_BIND: **d_bind** (arg);
…

Example: System Call Dependencies

fd = open (…)
*ioctl (fd, D_ALLOC, **arg1**)*
*ioctl (fd, D_BIND, **arg2**)*

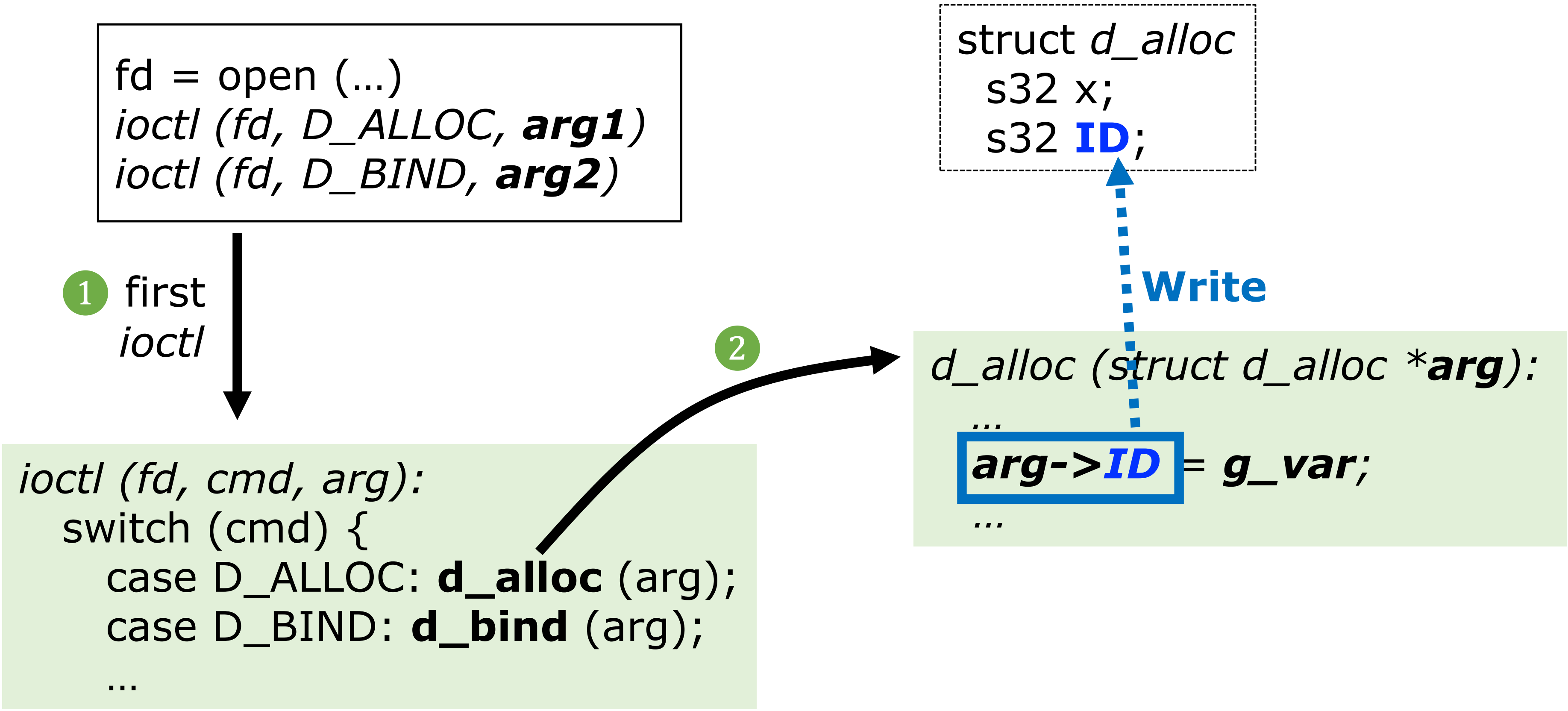
① first *ioctl*

ioctl (fd, cmd, arg):
switch (cmd) {
case D_ALLOC: **d_alloc** (arg);
case D_BIND: **d_bind** (arg);
…

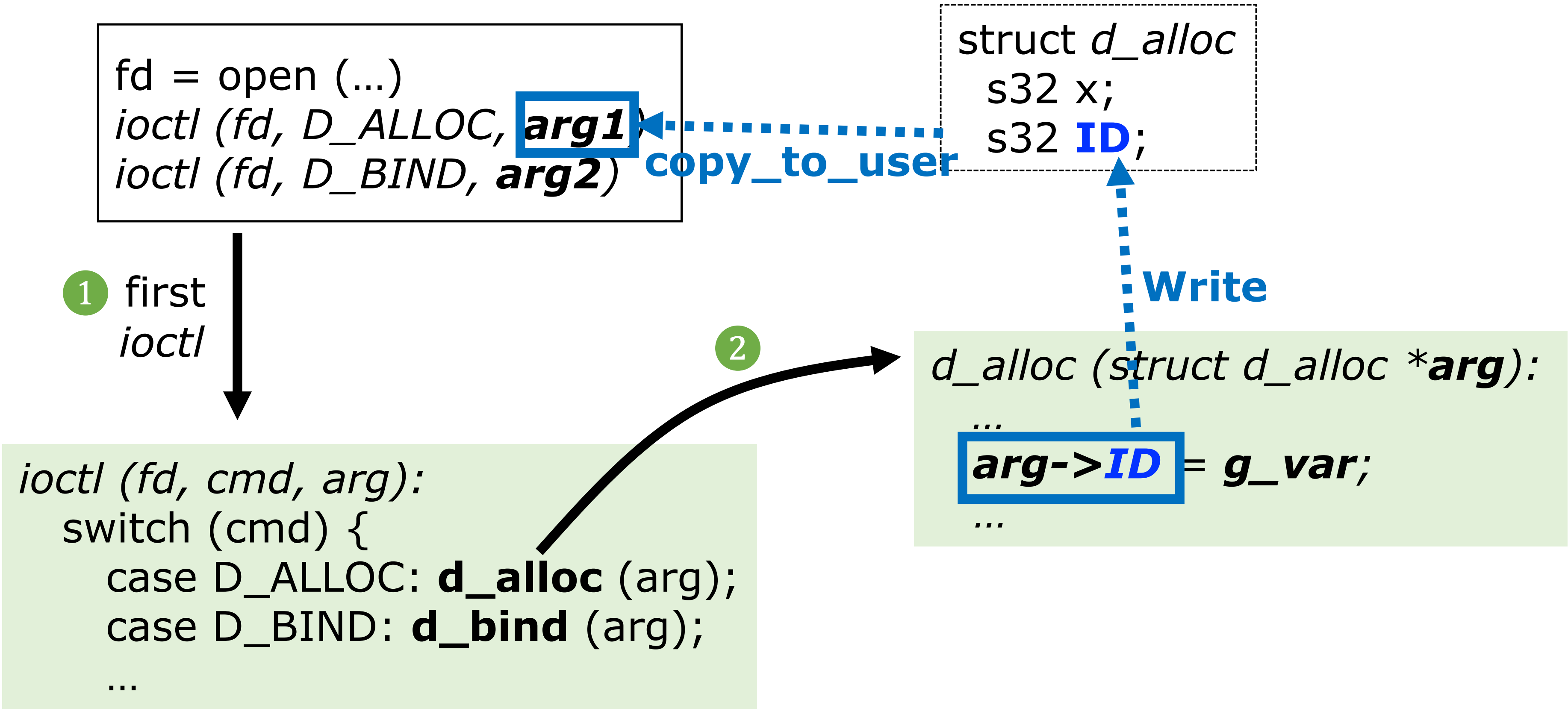
②

*d_alloc (struct d_alloc *__arg__):*
…
***arg->ID** = **g_var**;*
…

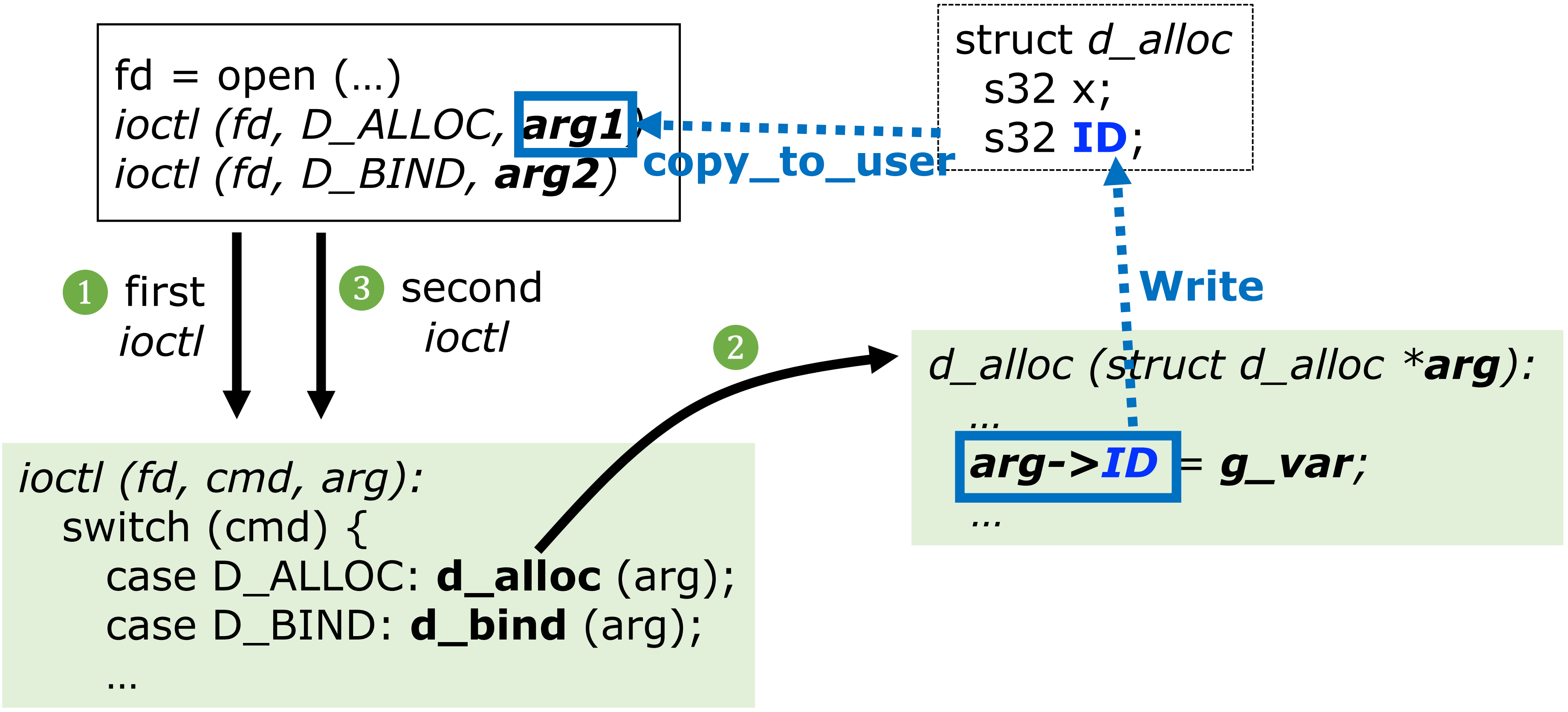
Example: System Call Dependencies



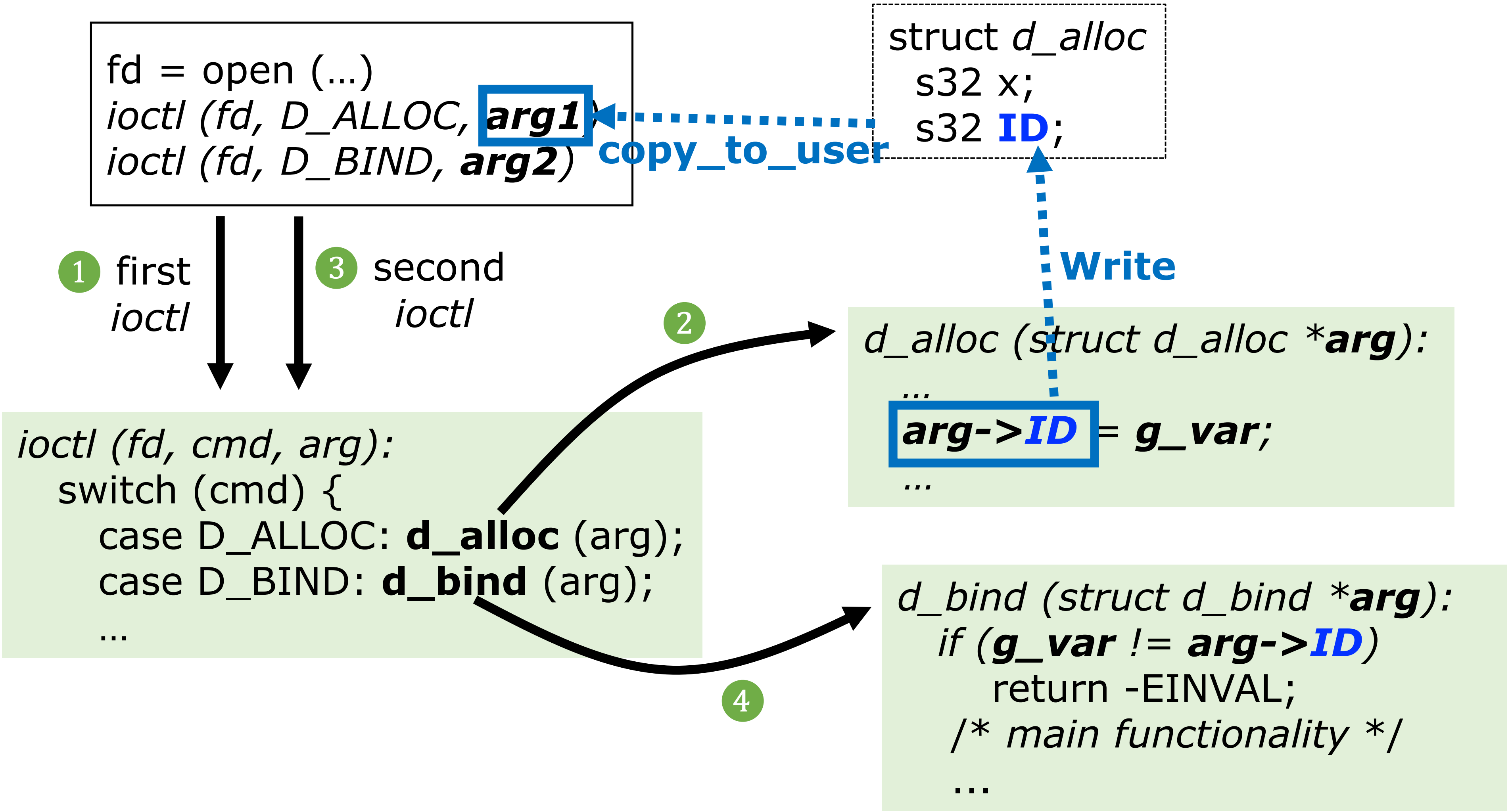
Example: System Call Dependencies



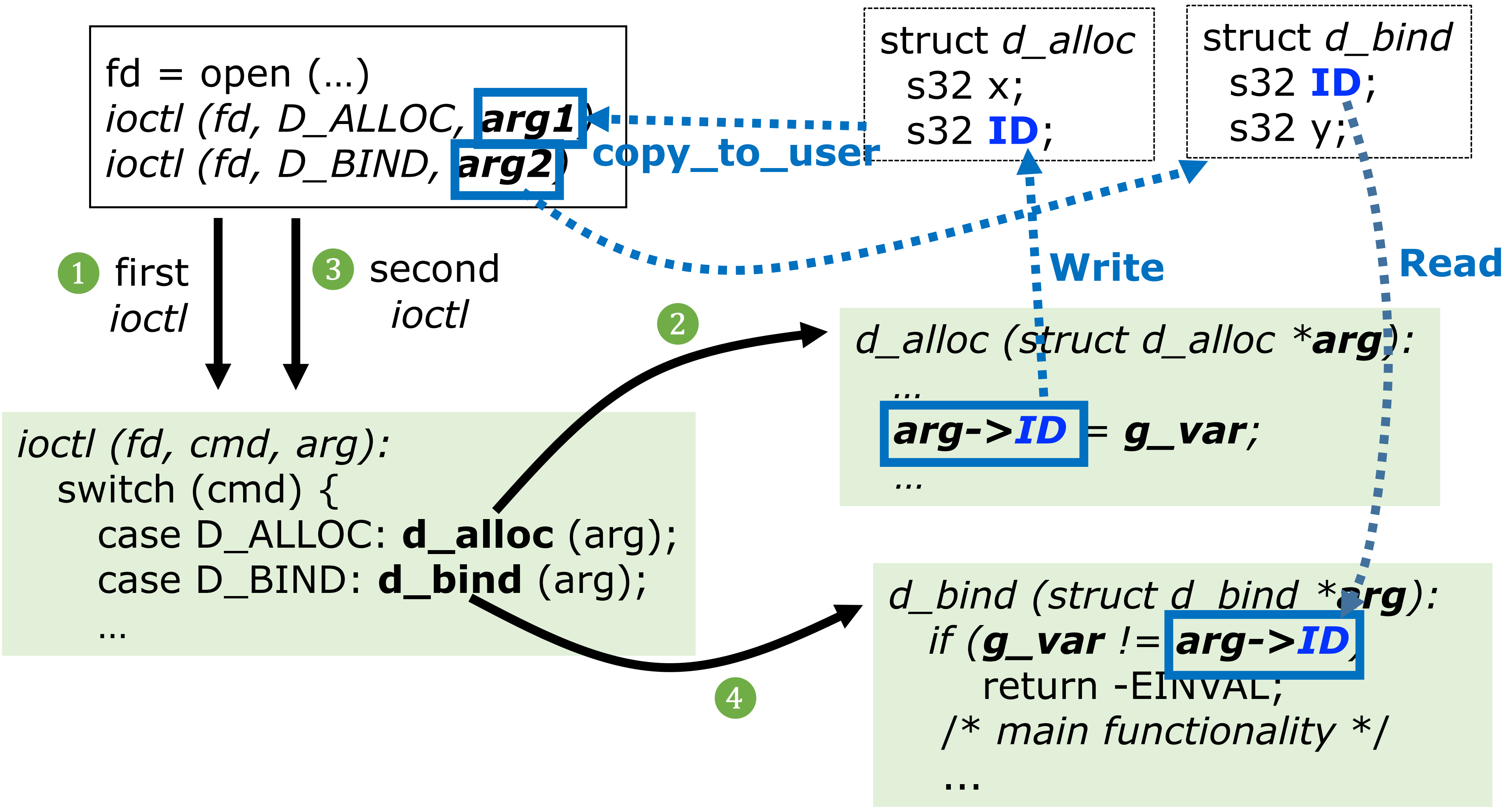
Example: System Call Dependencies



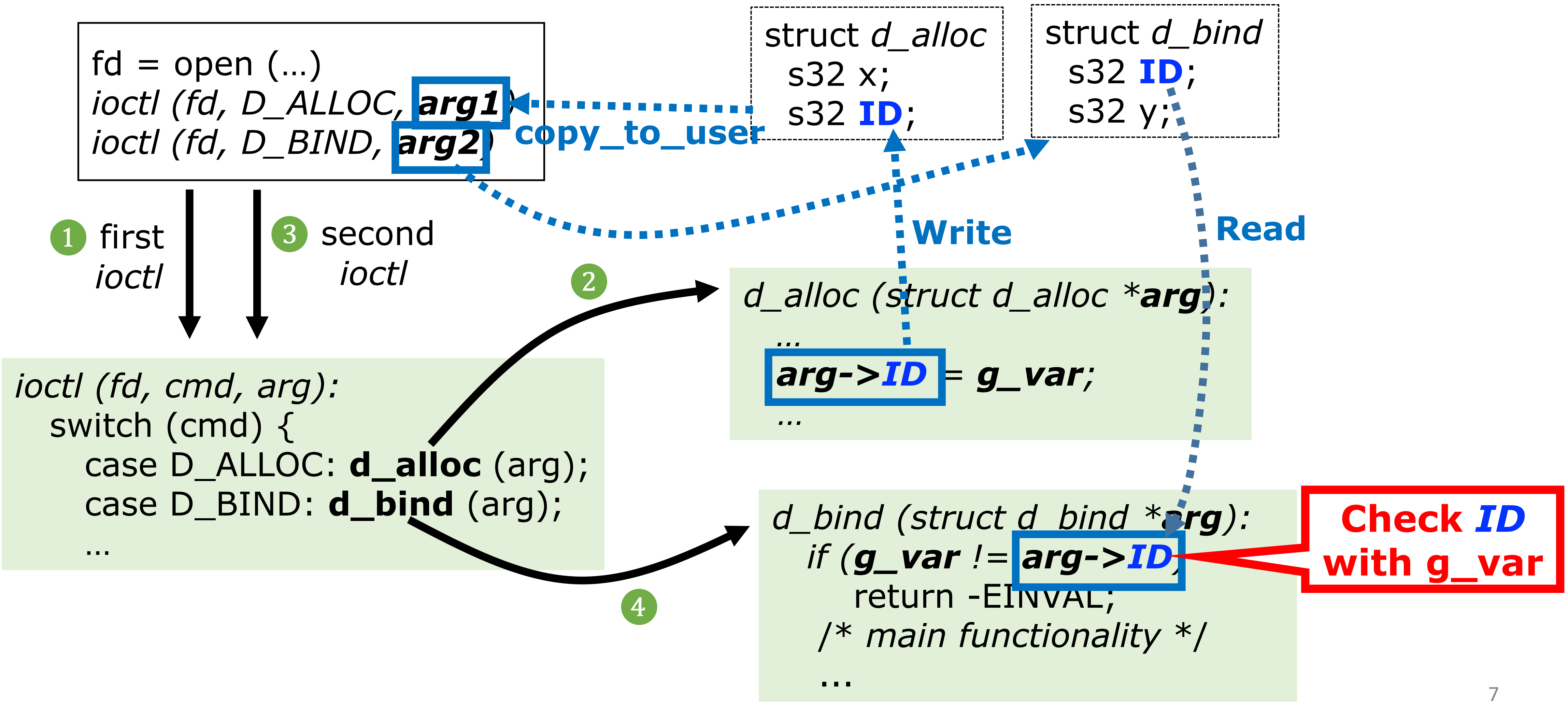
Example: System Call Dependencies



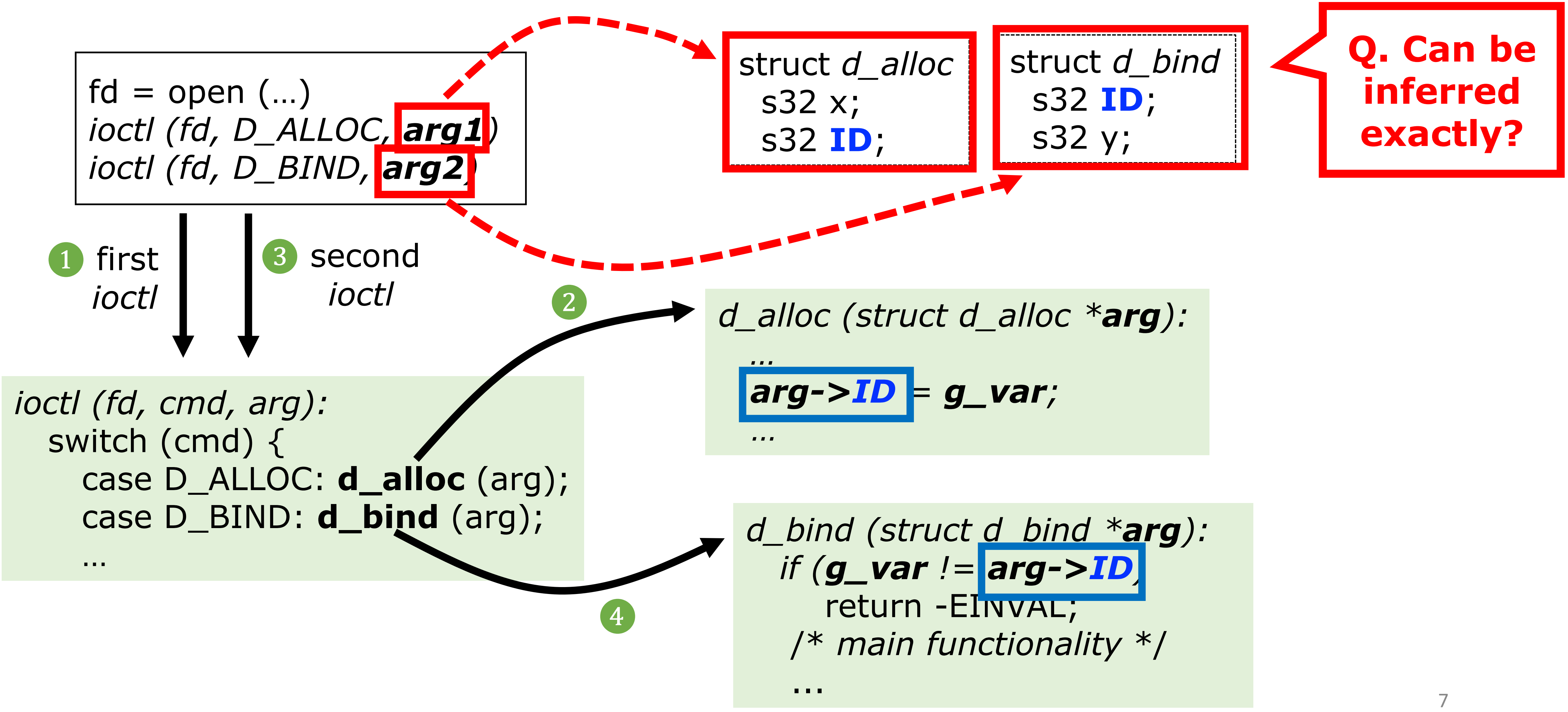
Example: System Call Dependencies



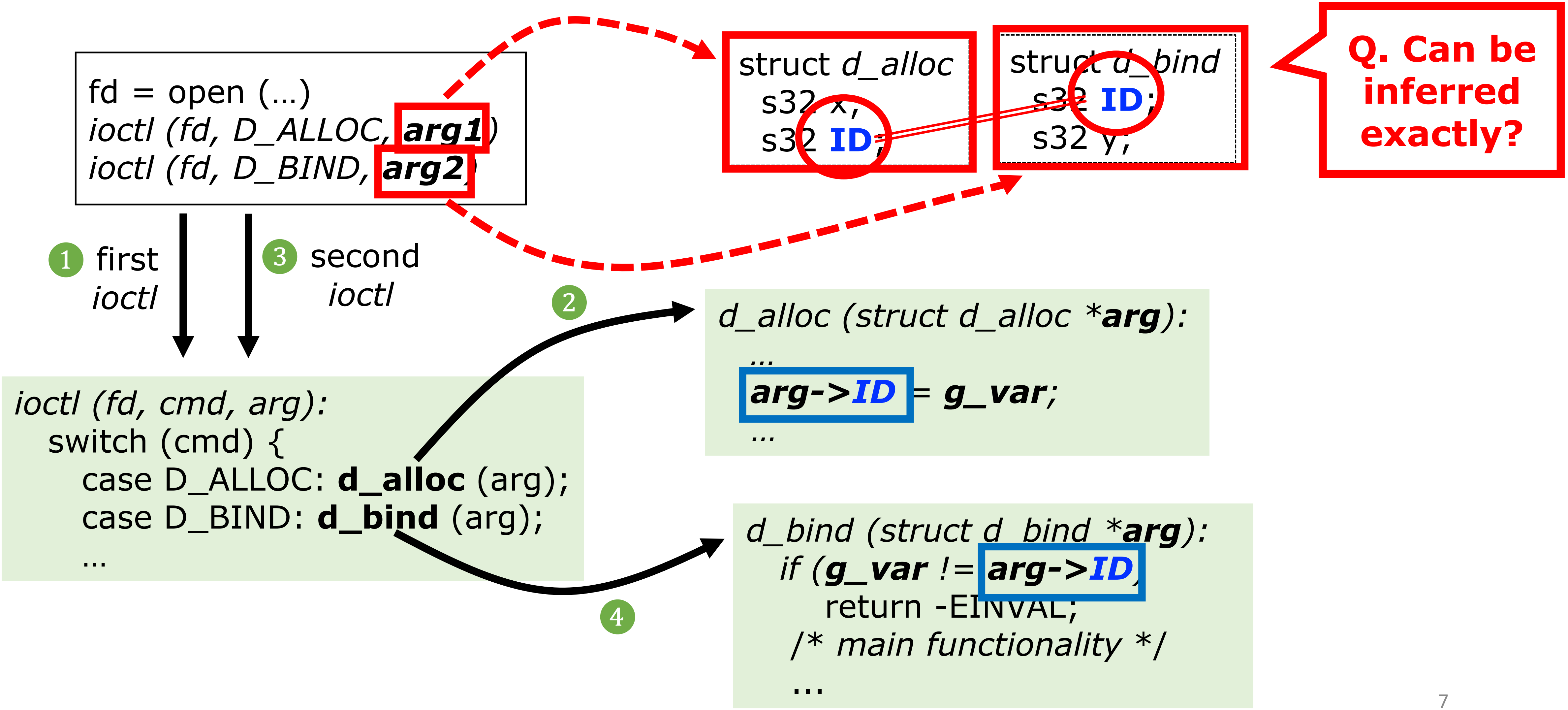
Example: System Call Dependencies



Example: System Call Dependencies



Example: System Call Dependencies



Challenge 3: Complex Argument Structure

ioctl (int fd, unsigned long cmd, ***void *argp****)*

write (int fd, ***void *buf****, size_t count)*

Challenge 3: Complex Argument Structure

unknown type

ioctl (int fd, unsigned long cmd, ***void *argp****)*

write (int fd, ***void *buf****, size_t count)*

unknown type

Example: Nested Arguments Structure

*ioctl (fd, USB_X, **arg**)*

Example: Nested Arguments Structure

ioctl (fd, USB_X, ***arg****)*

syscall

struct *usbdev_ctrl*:
void ****data***;
unsigned ***len***;

struct *usbdev_ctrl* ctrl;
uchar ****tbuf***;
…
copy_from_user *(&ctrl,* ***arg****, sizeof(ctrl))*
…
copy_from_user *(****tbuf****,* ***ctrl.data****,* ***ctrl.len****)*

/ do main functionality */*
…

Example: Nested Arguments Structure

ioctl (fd, USB_X, ***arg****)*

syscall

struct *usbdev_ctrl*:
    void ****data***;
    unsigned ***len***;

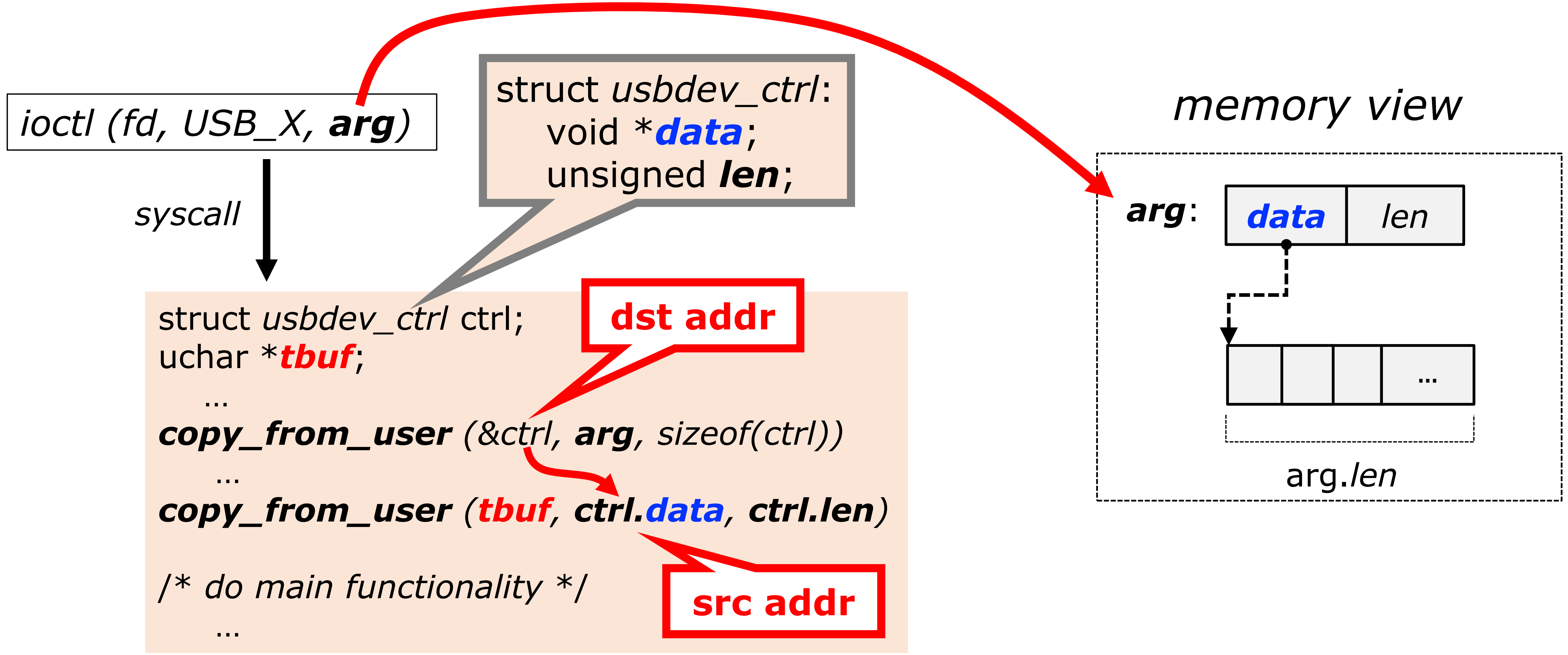
```
struct usbdev_ctrl ctrl;
uchar *tbuf;
    …
copy_from_user (&ctrl, arg, sizeof(ctrl))
    …
copy_from_user (tbuf, ctrl.data, ctrl.len)

/* do main functionality */
    …
```

dst addr

src addr

Example: Nested Arguments Structure



Example: Nested Arguments Structure

*ioctl (fd, USB_X, **arg**)*

syscall

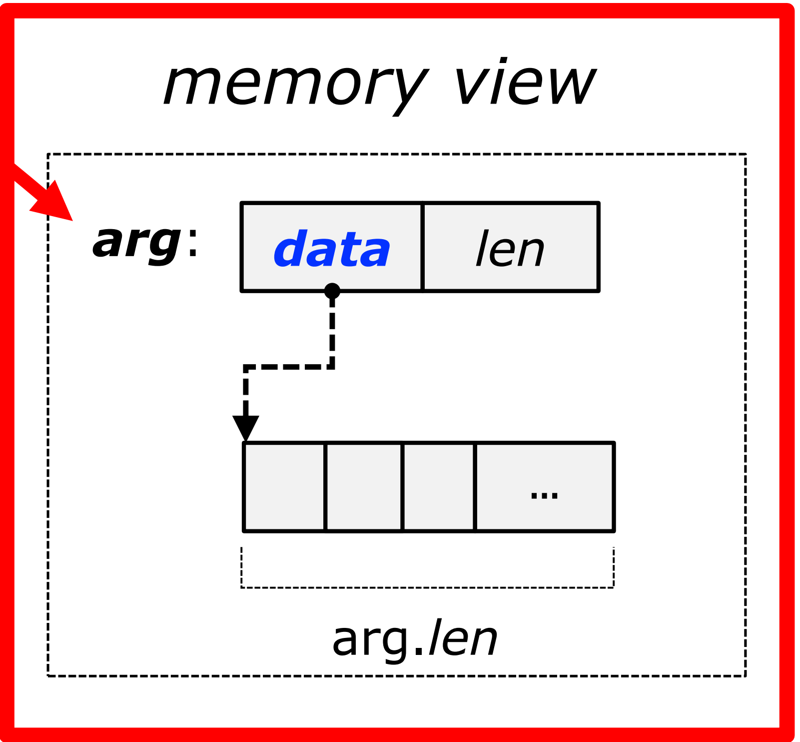
struct *usbdev_ctrl*:
    void ***data**;
    unsigned ***len***;

```
struct usbdev_ctrl ctrl;
uchar *tbuf;
    …
copy_from_user (&ctrl, arg, sizeof(ctrl))
    …
copy_from_user (tbuf, ctrl.data, ctrl.len)

/* do main functionality */
    …
```

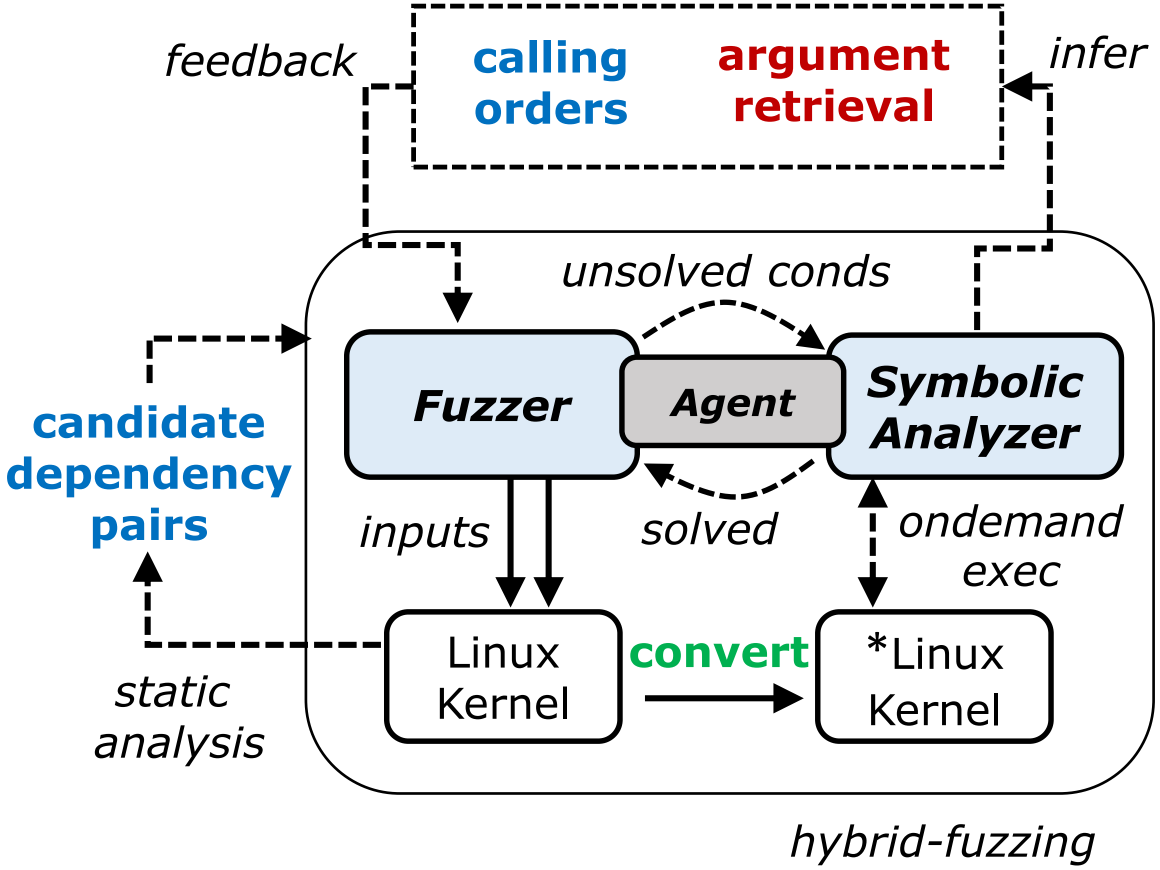
dst addr

src addr



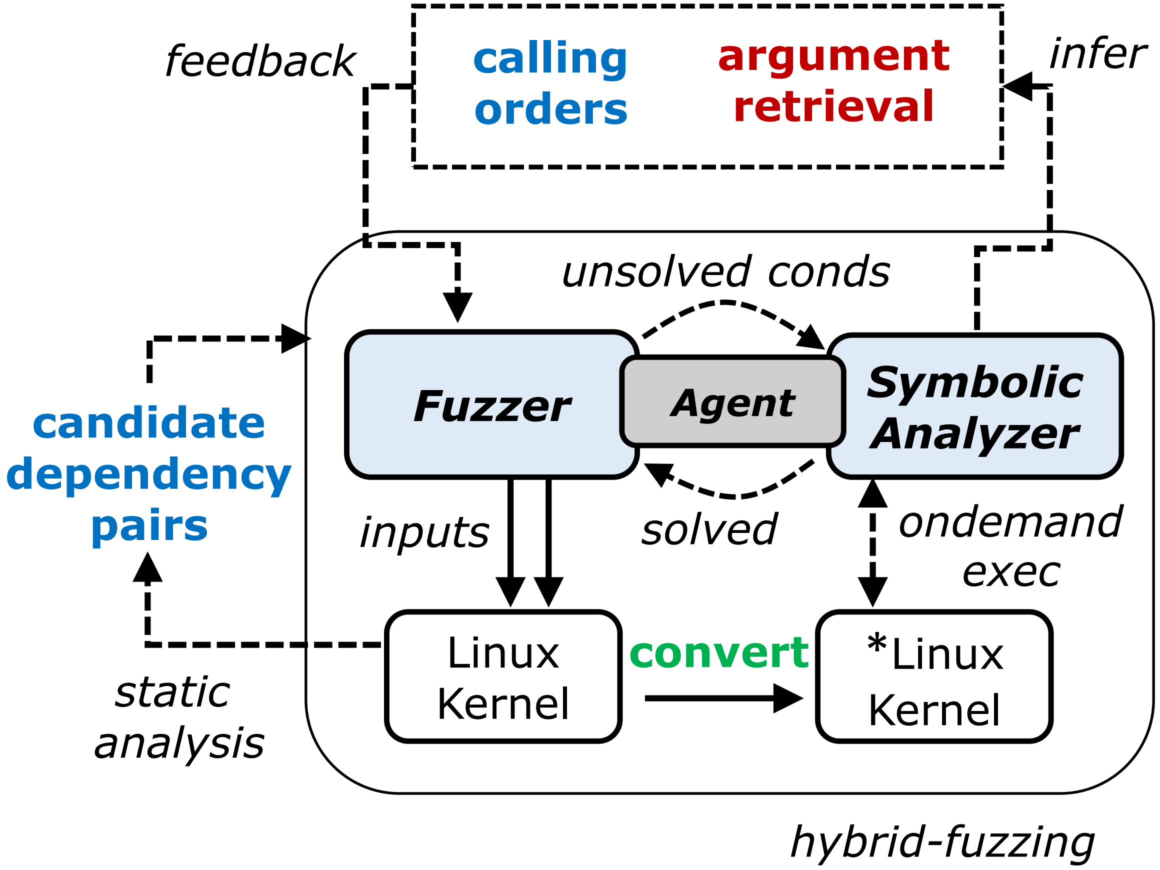
Q. Can be inferred exactly?

HFL: Hybrid Fuzzing on the Linux Kernel



- The *first* hybrid kernel fuzzer
- Coverage-guided/system call fuzzer
- Hybrid fuzzing
  - Combining *fuzzer* and *symbolic analyzer*
  - *Agent* act as a glue between the two components

HFL: Hybrid Fuzzing on the Linux Kernel



- Handling the challenges

1. *Implicit control transfer*
   - ***Convert to direct control-flow***
2. *System call dependencies*
   - ***Infer system call dependency***
3. *Complex argument structure*
   - ***Infer nested argument structure***

1. Conversion to Direct Control-flow

<Before>

```
idx = cmd – INFO_FIRST;
    ...
funp = _ioctls[idx];
    ...
funp (sbi, param);
```

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
        ...
    ioctl_ismountpoint,
};
```

1. Conversion to Direct Control-flow

<Before>

```
idx = cmd – INFO_FIRST;
    …
funp = _ioctls[idx];
    …
funp (sbi, param);
```

Compile time conversion:
direct **control transfer**

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
        …
    ioctl_ismountpoint,
};
```

functions

<After>

```
idx = cmd – INFO_FIRST;
    …
funp = _ioctls[idx];
    …
if (cmd == IOCTL_VERSION)
      ioctl_version (sbi, param);
else if (cmd == IOCTL_PROTO)
      ioctl_protover (sbi, param);
  …
      ioctl_ismountpoint (sbi, param)
```

2. Syscall Dependency Inference

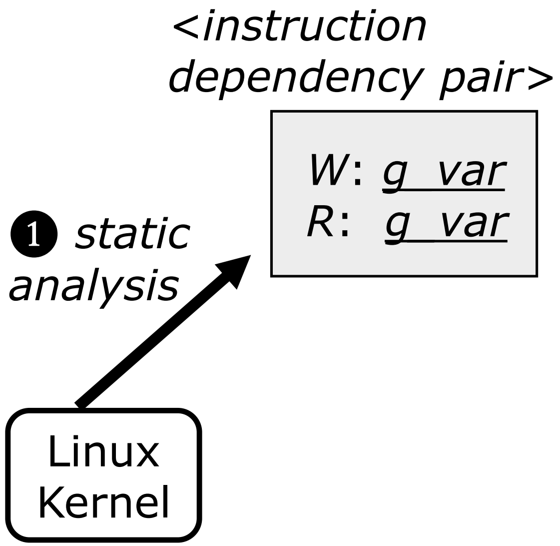
1. *Collecting W-R pairs*
2. *Runtime validation*
3. *Parameter dependency*

fd = open (…)
ioctl (fd, D_ALLOC, ***{struct d_alloc}****)*
ioctl (fd, D_BIND, ***{struct d_bind}****)*

2. Syscall Dependency Inference

1. *Collecting W-R pairs*
2. *Runtime validation*
3. *Parameter dependency*

```
fd = open (…)
ioctl (fd, D_ALLOC, {struct d_alloc})
ioctl (fd, D_BIND,  {struct d_bind})
```



2. Syscall Dependency Inference

❶ *Collecting W-R pairs*

❷ *Runtime validation*

❸ *Parameter dependency*

```
fd = open (…)
ioctl (fd, D_ALLOC, {struct d_alloc})
ioctl (fd, D_BIND, {struct d_bind})
```

symbolize

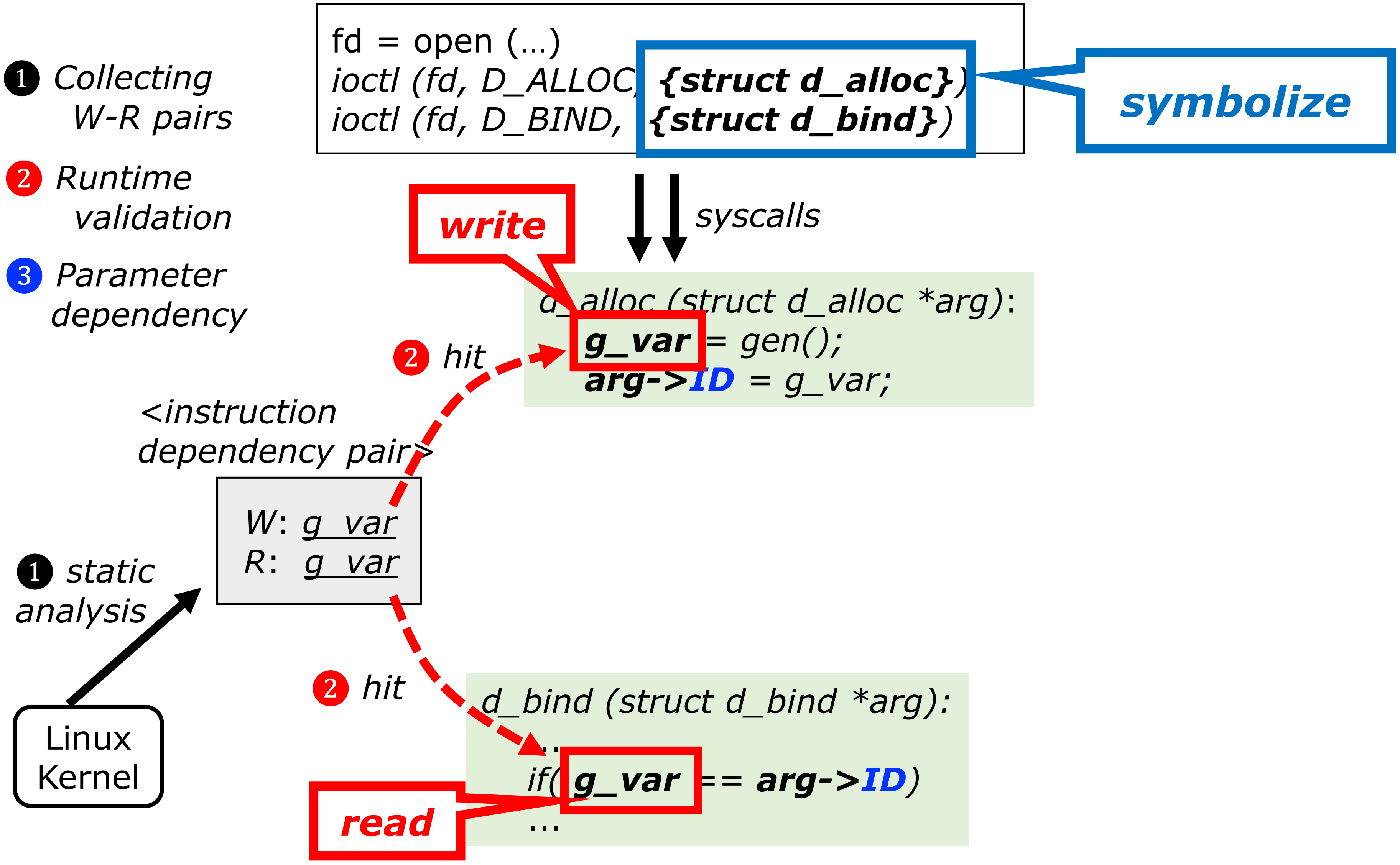
<instruction dependency pair>

W: g_var
R: g_var

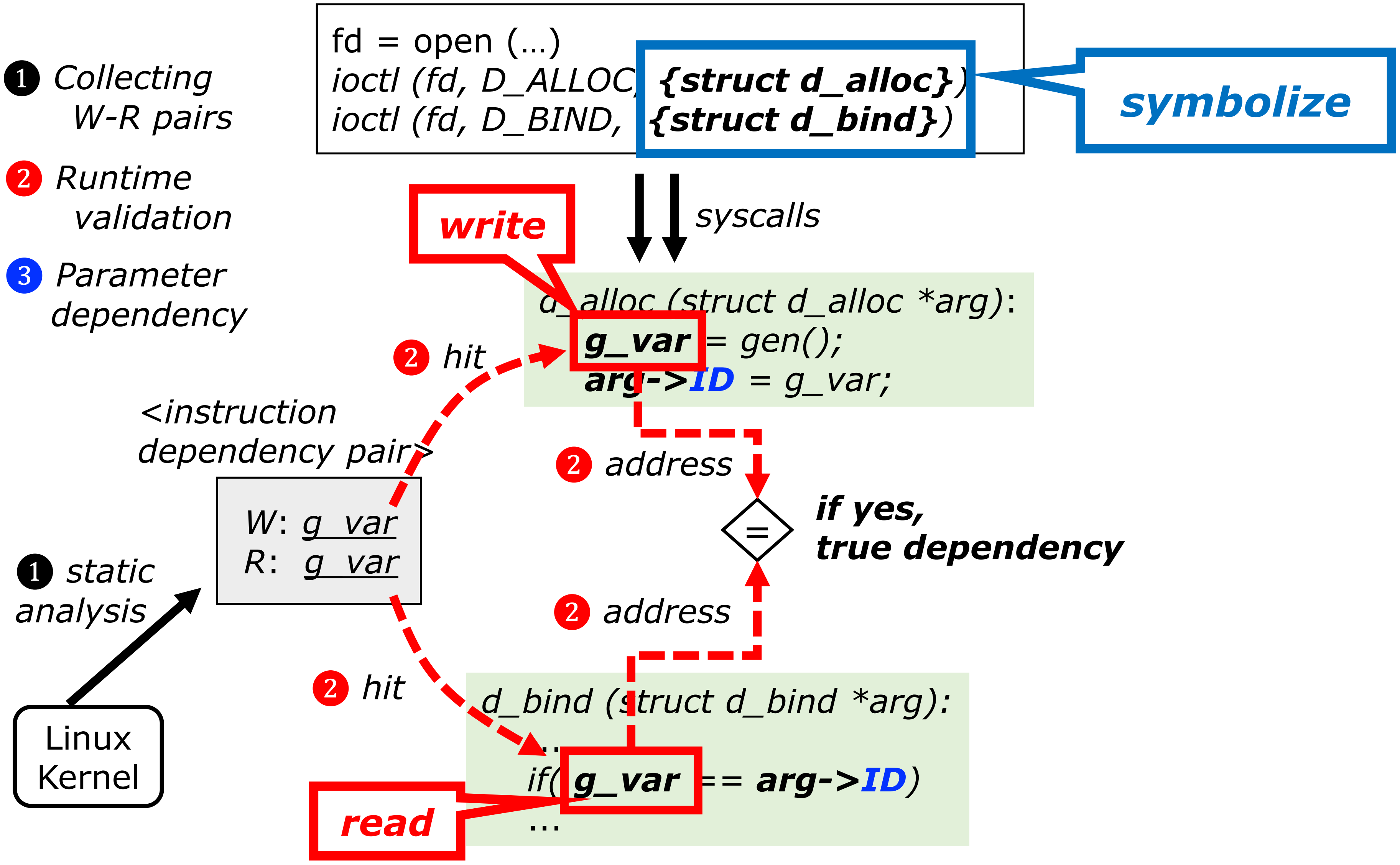
❶ *static analysis*

Linux Kernel

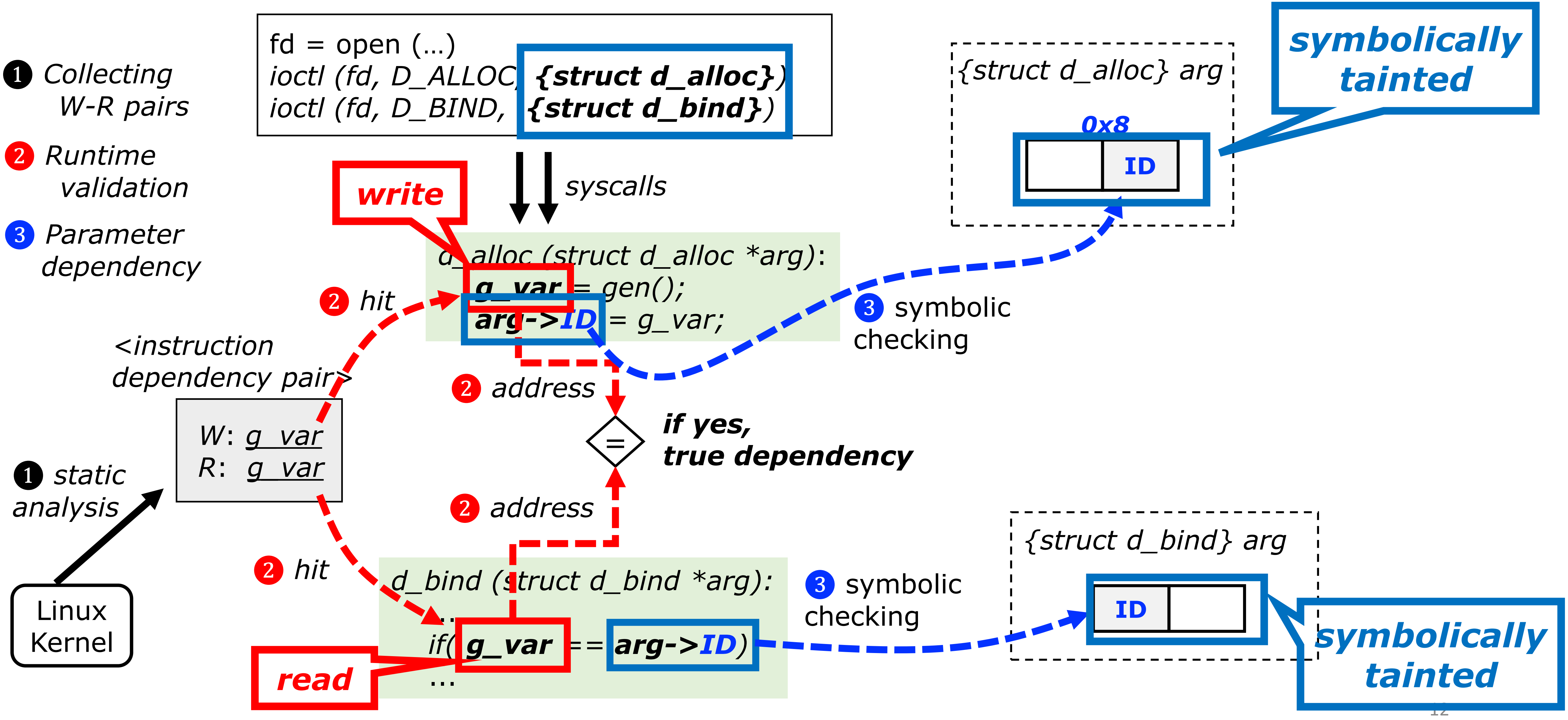
2. Syscall Dependency Inference



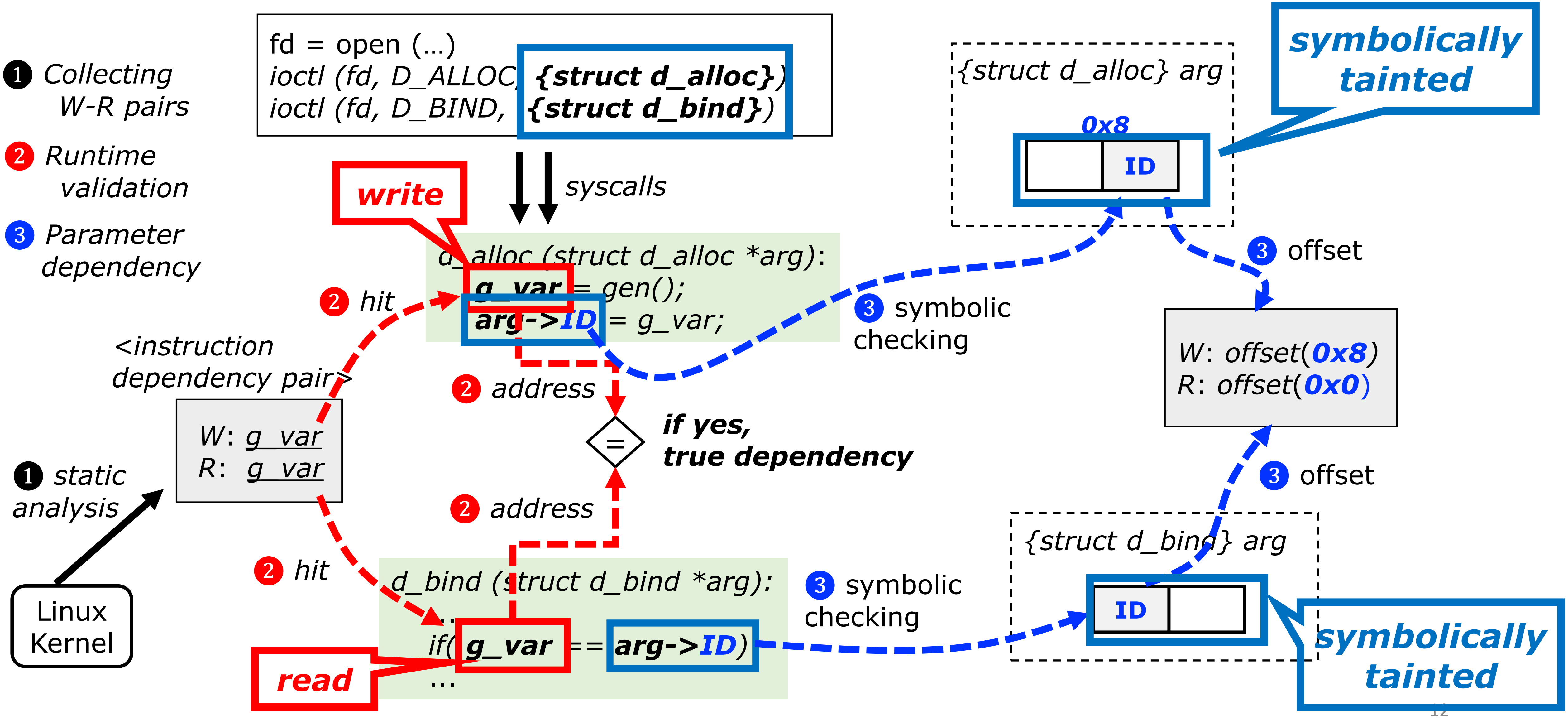
2. Syscall Dependency Inference



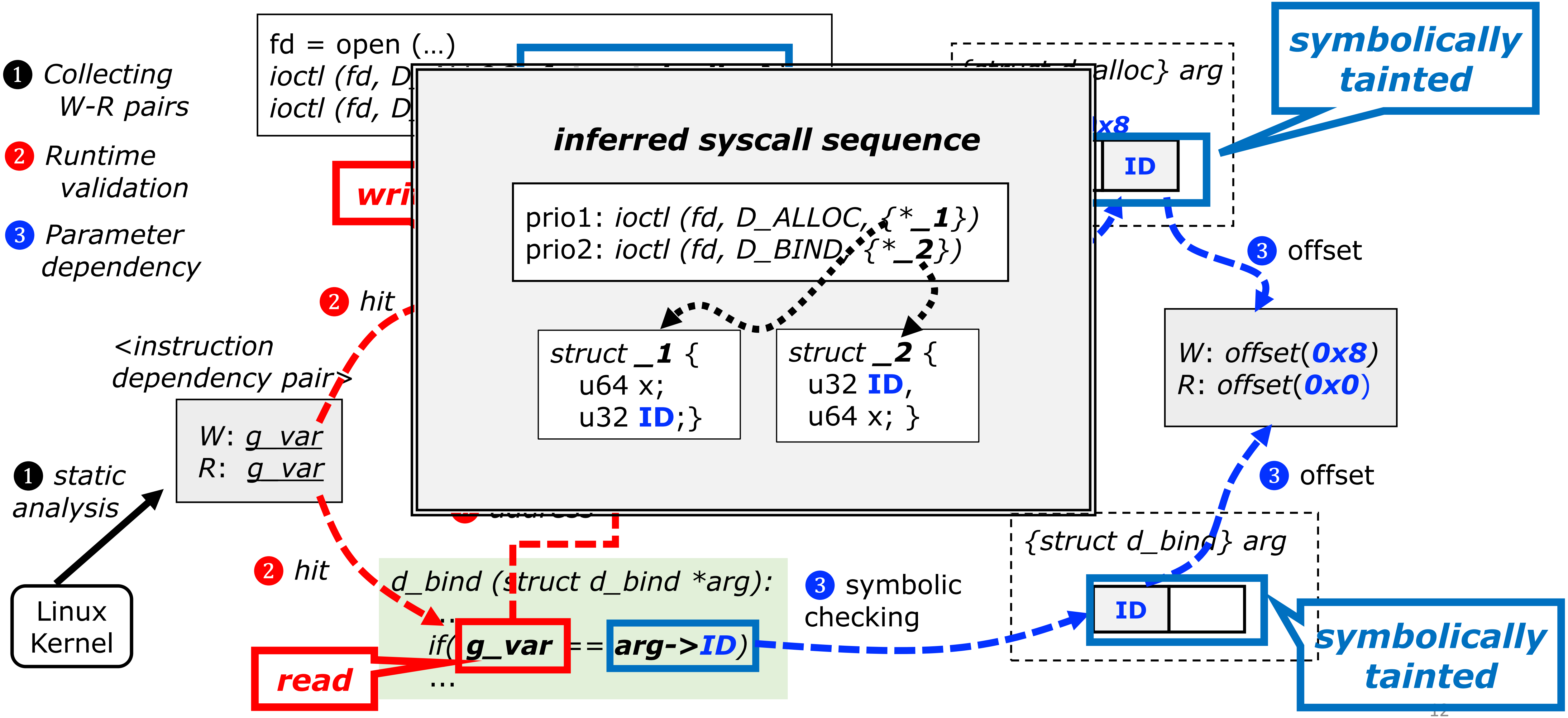
2. Syscall Dependency Inference



2. Syscall Dependency Inference



2. Syscall Dependency Inference



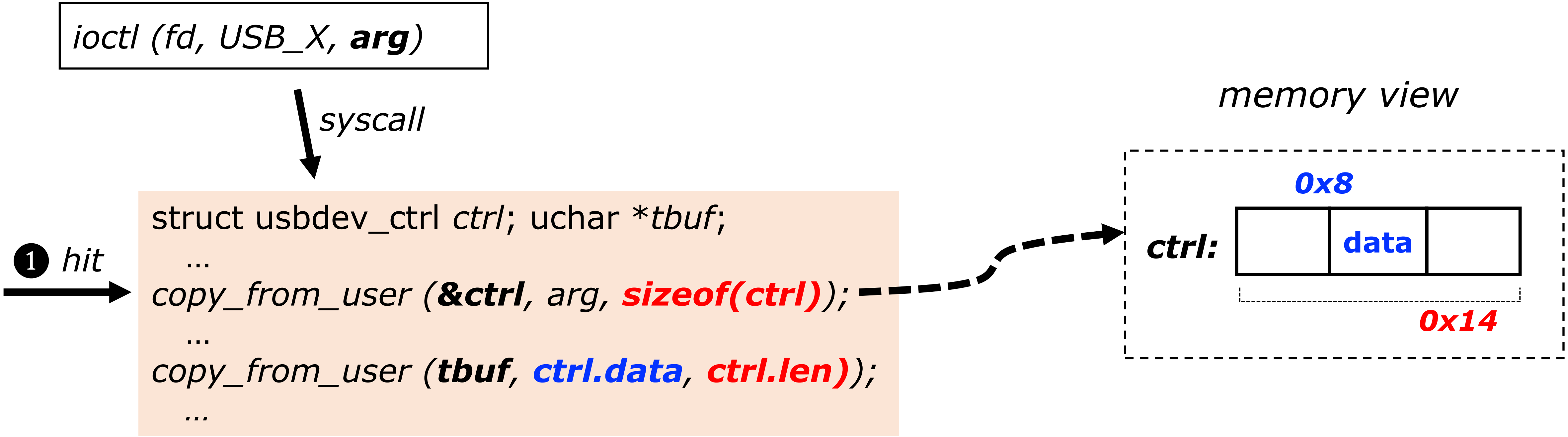
3. Nested Argument Format Retrieval

ioctl (fd, USB_X, ***arg****)*

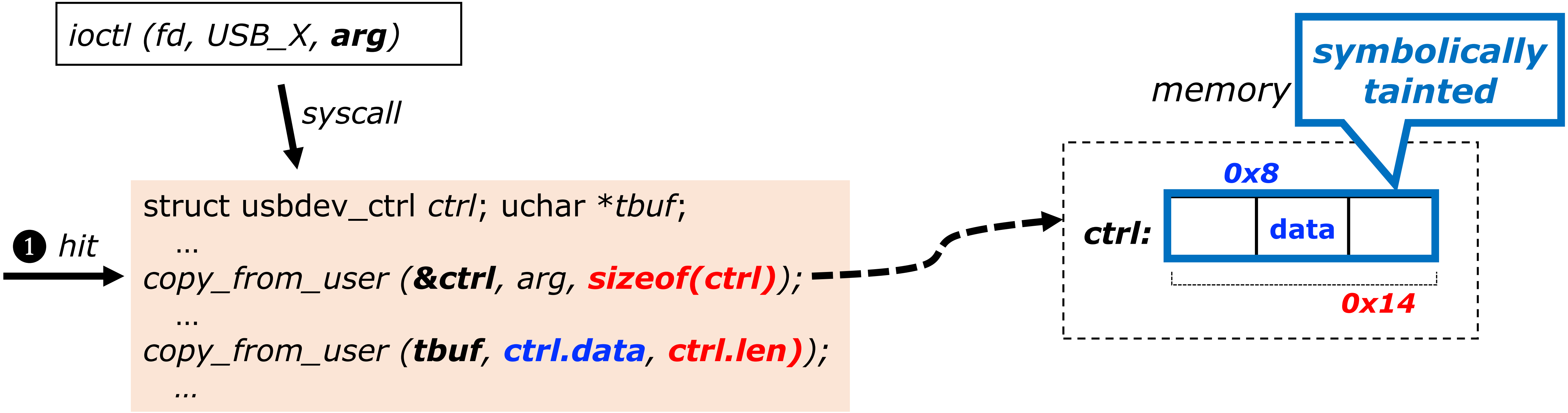
syscall

struct usbdev_ctrl *ctrl*; uchar **tbuf*;
...
*copy_from_user (**&ctrl**, arg, **sizeof(ctrl)**);*
...
*copy_from_user (**tbuf**, **ctrl.data**, **ctrl.len)**);*
...

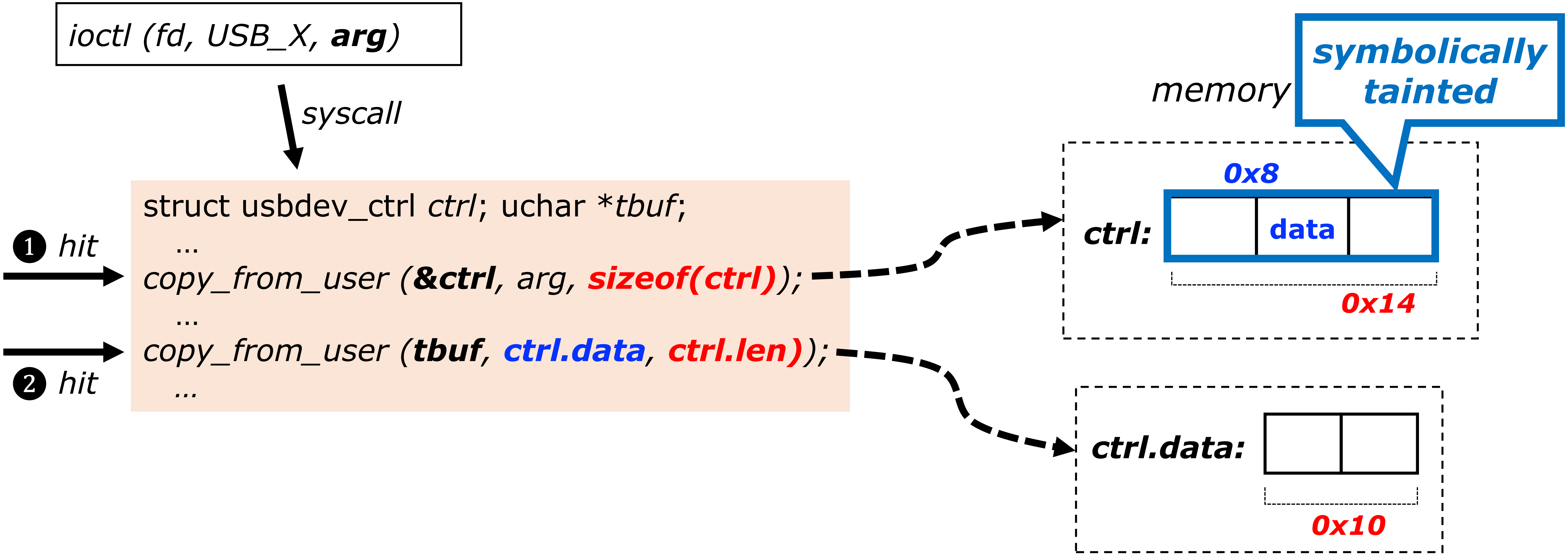
3. Nested Argument Format Retrieval



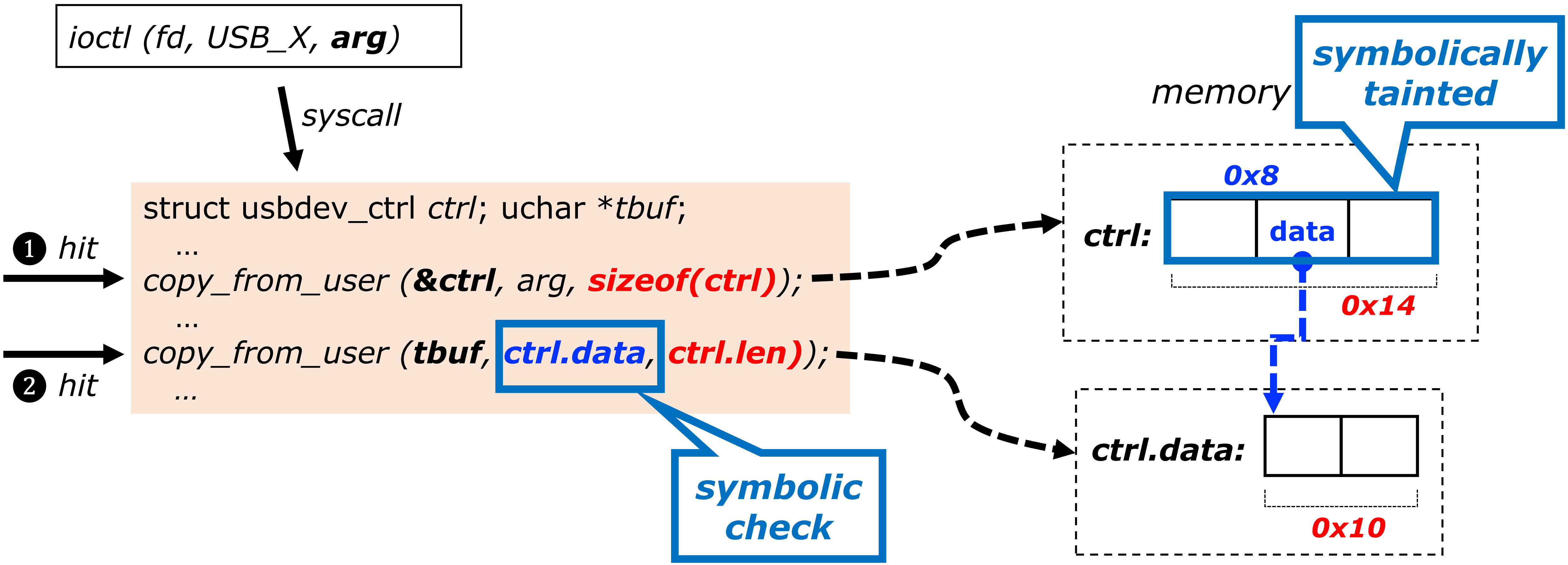
3. Nested Argument Format Retrieval



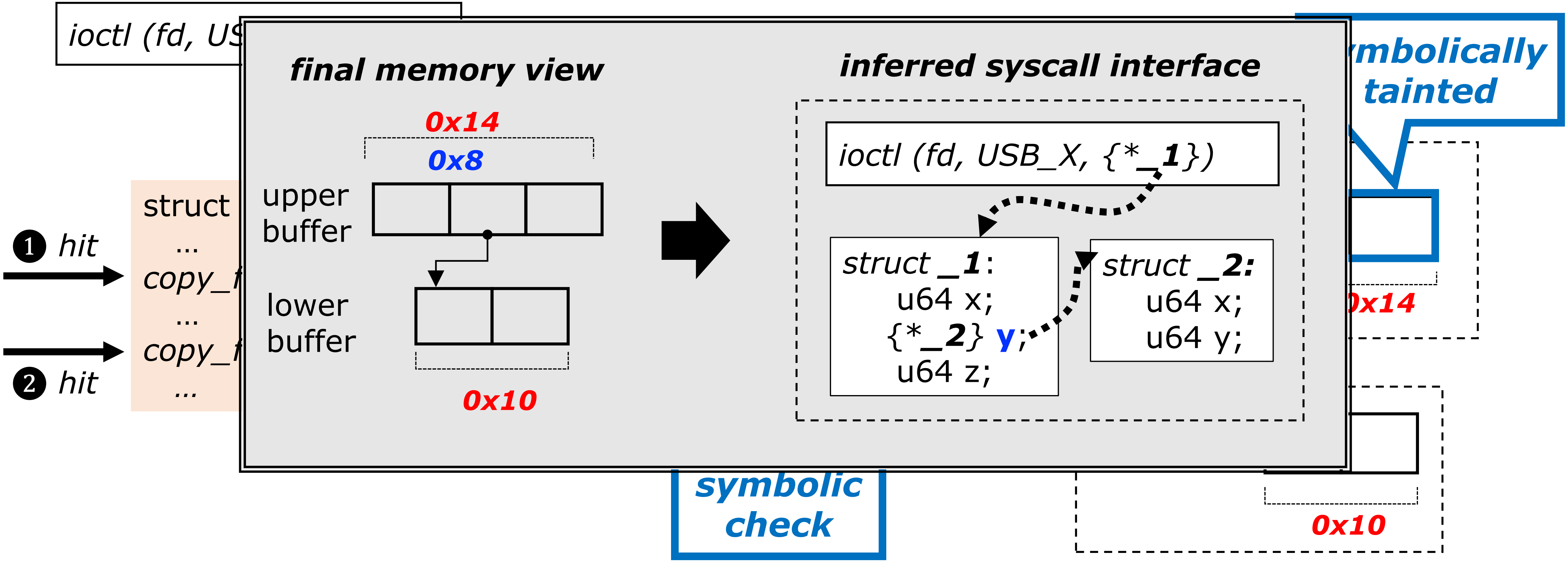
3. Nested Argument Format Retrieval



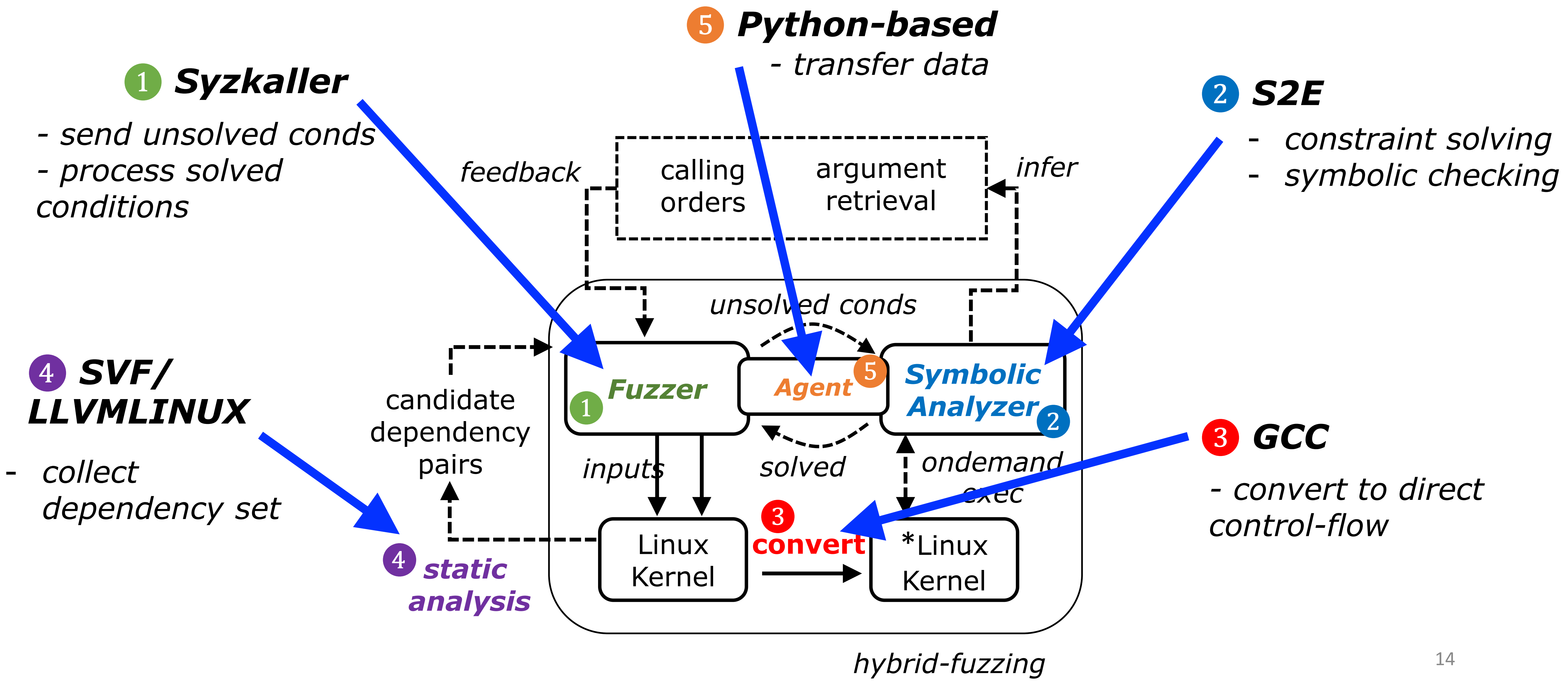
3. Nested Argument Format Retrieval



3. Nested Argument Format Retrieval



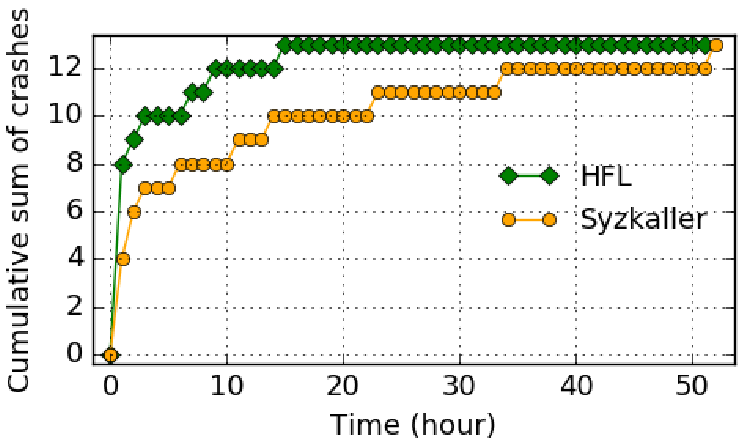
Implementation



Vulnerability Discovery

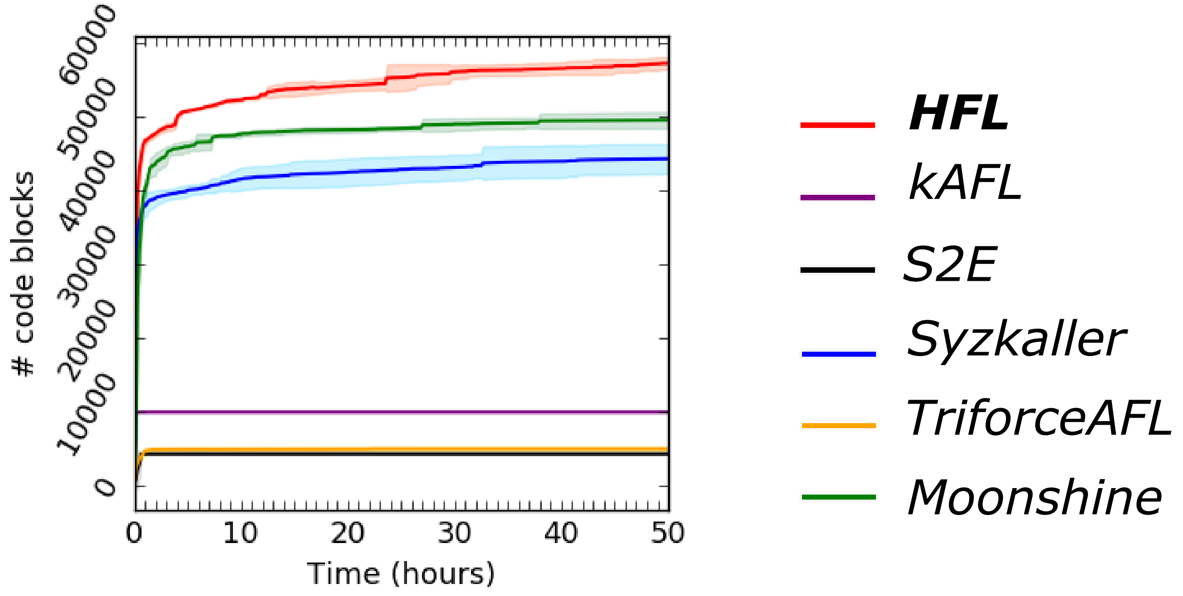
- Discovered new vulnerabilities
  - ***24 new vulnerabilities*** found in the Linux kernels
    - 17 confirmed by Linux kernel community
  - UAF, integer overflow, uninitialized variable access, etc.

- Efficiency of bug-finding capability
  - 13 known bugs for HFL and Syzkaller
  - They were all found by HFL ***3x*** faster than Syzkaller

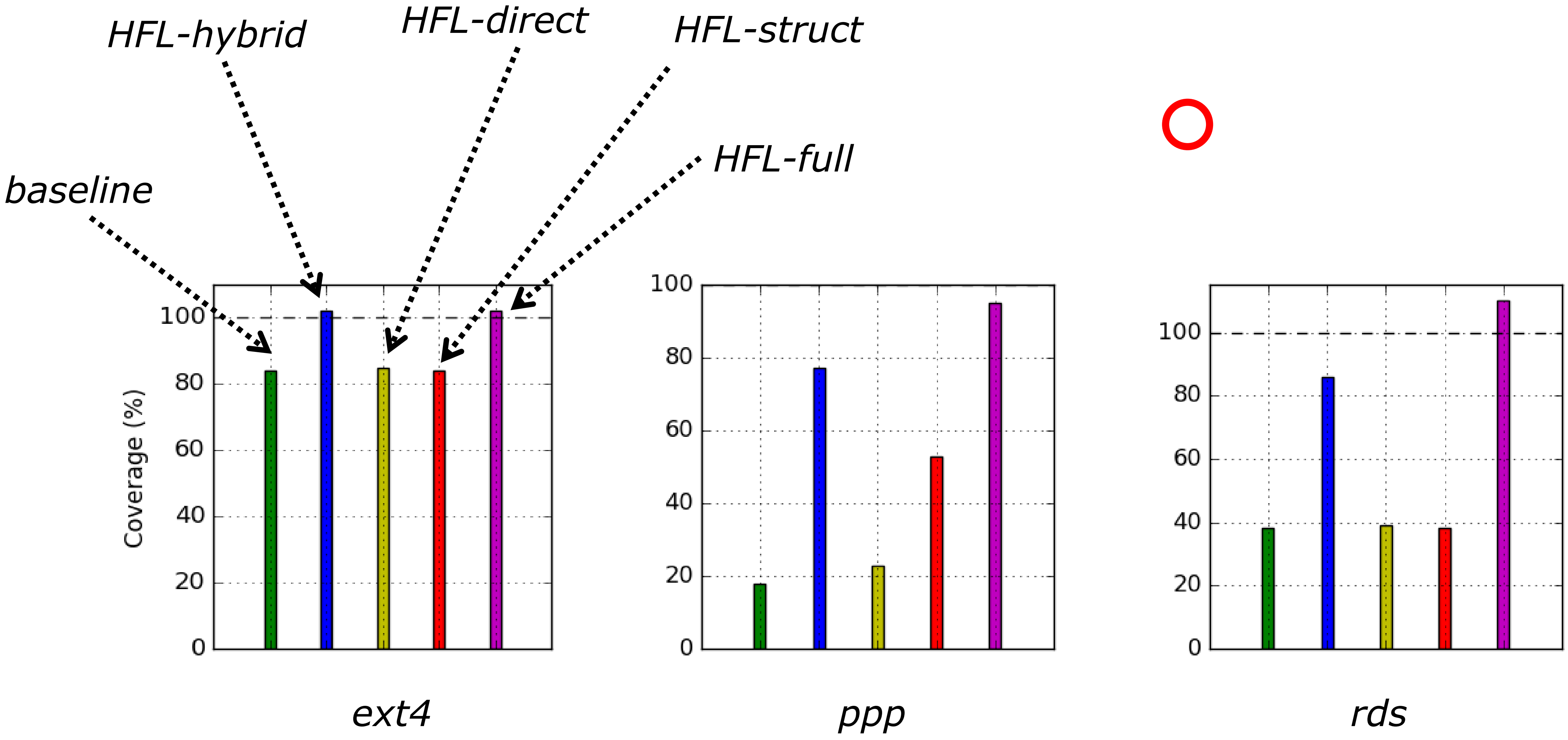


Code Coverage Enhancement

- Compared with state-of-the-art kernel fuzzers
  - *Moonshine [Sec'18], kAFL [CCS'17], etc.*
- *KCOV*-based coverage measurement
- HFL presents coverage improvement over the others
  - *Ranging from **15%** to **4x***



Effectiveness of HFL per-feature solution



Conclusion

- HFL is the *first* hybrid kernel fuzzer.
- HFL addresses the crucial challenges in the Linux kernel.
- HFL found 24 new vulnerabilities, and presented the better code coverage, compared to state-of-the-arts.

Thank you