HFL: Hybrid Fuzzing on the Linux Kernel

Kyungtae Kim*, Dae R. Jeong°, Chung Hwan Kim¶, Yeongjin Jang§, Insik Shin°, Byoungyoung Lee¹*

*Purdue University, °KAIST, ¶NEC Labs America, §Oregon State University, ¹Seoul National University
Software Security Analysis

• Random fuzzing
  • **Pros**: Fast path exploration
  • **Cons**: Strong branch conditions e.g., \( if(i == 0x\text{deadbeef}) \)

• 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?

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]
Challenge 1: Indirect Control Transfer

\[
\text{idx} = \mathbf{cmd} - \text{INFO\_FIRST};
\]
\[
\ldots
\]
\[
\text{funp} = \_\text{ioctl}[\text{idx}];
\]
\[
\ldots
\]
\[
\text{funp} (\text{sbi}, \text{param});
\]

\[<\text{indirect function call}>\]

\[
\text{ioctl\_fn} \_\text{ioctl[]}[] = \{
\text{ioctl\_version},
\text{ioctl\_protover},
\ldots
\text{ioctl\_ismountpoint},
\};
\]

\[<\text{function pointer table}>\]
Challenge 1: Indirect Control Transfer

```
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>

Derived from syscall arguments
Challenge 1: Indirect Control Transfer

\[
\text{ioctl\_fn\_ioctl\_s[] = \{}
\begin{align*}
\text{ioctl\_version,} \\
\text{ioctl\_protover,} \\
\text{...} \\
\text{ioctl\_ismountpoint,}
\end{align*}
\]

\[
\text{funp (sbi, param)};
\]

\[
\text{idx = cmd - INFO\_FIRST;}
\]

\[
\begin{align*}
\text{...} \\
\text{funp = \_ioctl\_s[idx];} \\
\text{...}
\end{align*}
\]

\[
\text{<indirect function call>}
\]

\[
\text{<function pointer table>}
\]

derived from syscall arguments
Challenge 1: Indirect Control Transfer

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

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

<indirect function call>

derived from syscall arguments

indirect control transfer
Challenge 1: Indirect Control Transfer

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

Derived from syscall arguments

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

Targets to be hit

*indirect function call*

*indirect control transfer*
Challenge 1: Indirect Control Transfer

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

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

<indirect function call>

derived from syscall arguments

targets to be hit

indirect control transfer
Challenge 1: Indirect Control Transfer

Q. Can be fuzzed enough to explore all functions?

\[
\text{idx} = \text{cmd} - \text{INFO\_FIRST};
\]

\[
\ldots
\]

\[
\text{funp} = \_\text{iocls}[\text{idx}];
\]

\[
\ldots
\]

\[
\text{funp}(\text{sbi}, \text{param});
\]

\[
<\text{indirect function call}>
\]

\[
\text{targets to be hit}
\]

\[
\text{iocls[]} = \{
\text{iocltl\_version, iocltl\_protover,}
\ldots
\text{iocltl\_ismountpoint,}
\}
\]
Challenge 2: System Call Dependencies

\[
\begin{align*}
\text{int } & \text{open} \quad \text{(const char } *\text{pathname, int flags, mode}_t \text{ mode)} \\
\text{ssize_t } & \text{write} \quad \text{(int } \text{fd, void } *\text{buf, size}_t \text{ count)} \\
\text{ioctl} & \quad \text{(int } \text{fd, unsigned long req, void } *\text{argp)} \\
\text{ioctl} & \quad \text{(int } \text{fd, unsigned long req, void } *\text{argp)}
\end{align*}
\]
Challenge 2: System Call Dependencies

**explicit syscall dependencies**

\[
\begin{align*}
\textbf{int open} & \quad (\text{const char *pathname, int flags, mode\_t mode}) \\
\textbf{ssize\_t write} & \quad (\textbf{int fd}, \text{void *buf, size\_t count}) \\
\textbf{ioctl} & \quad (\text{int fd, unsigned long req, void *argp}) \\
\textbf{ioctl} & \quad (\text{int fd, unsigned long req, void *argp})
\end{align*}
\]
Challenge 2: System Call Dependencies

explicit syscall dependencies

\[
\begin{align*}
\text{int open} & \quad \text{(const char *pathname, int flags, mode_t mode)} \\
\text{ssize_t write} & \quad \text{(int fd, void *buf, size_t count)} \\
\text{ioctl} & \quad \text{(int fd, unsigned long req, void *argp)} \\
\text{ioctl} & \quad \text{(int fd, unsigned long req, void *argp)}
\end{align*}
\]

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_ALLOCC, arg1)  
ioctl (fd, D_BIND, arg2)
```

1 first ioctl

```
ioctl (fd, cmd, arg):
    switch (cmd) {
        case D_ALLOCC: 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)

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

1 first ioctl
2 d_alloc (struct d_alloc *arg):
    ...
    arg->ID = g_var;
    ...

Example: System Call Dependencies

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

```c
struct d_alloc
  s32 x;
s32 ID;
```

```c
ioctl (fd, cmd, arg):
  switch (cmd) {
    case D_ALLOC: d_alloc (arg);
    case D_BIND: d_bind (arg);
    ...
```

```c
Write
```

```c
d_alloc (struct d_alloc *arg):
  ...
  arg->ID = g_var;
  ...
```
Example: System Call Dependencies

fd = open (...)  
ioctl (fd, D_ALLOC, arg1)  
ioctl (fd, D_BIND, arg2)

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

struct d_alloc
  s32 x;
  s32 ID;

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

Copy to User

Write
Example: System Call Dependencies

```c
fd = open (...)  
ioctl (fd, DALLOC, arg1)  
ioctl (fd, DBIND, arg2)
```

1. **first ioctl**
2. **second ioctl**
3. **ioctl (fd, cmd, arg):**
   ```c
   switch (cmd) {  
   case DALLOC:  
     d_alloc (arg);  
   case DBIND:  
     d_bind (arg);  
   ...  
   }
   ```

---

```c
struct d_alloc  
  s32 x;  
  s32 ID;  
writing
```

- **Write**
- **copy_to_user**
- **first ioctl**
- **second ioctl**
- **d_alloc (struct d_alloc *arg):**
  ```c
  ...  
  arg->ID = g_var;  
  ...  
  ```
Example: System Call Dependencies

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

```
struct d_alloc  
s32 x;  
s32 ID;
```

```
d_alloc (struct d_alloc *arg):  
...  
arg->ID = g_var;  
...
```

```
d_bind (struct d_bind *arg):  
if (g_var != arg->ID)  
return -EINVAL;  
/* main functionality */  
...  
```

1. first ioctl
2. Write
3. second ioctl
4. copy_to_user

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

- if (g_var != arg->ID)
  - return -EINVAL;
  - /* main functionality */

- 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)

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

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

d_bind (struct d_bind *arg):
if (g_var != arg->ID)
    return -EINVAL;
/* main functionality */
...

struct d_alloc
s32 x;
s32 ID;

struct d_bind
s32 ID;
s32 y;

first ioctl

second ioctl

write

read

Read

Write

1st ioctl

2nd ioctl

Copy to user

arg1

arg2
Example: System Call Dependencies

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

struct d_alloc
  s32 x;
  s32 ID;

struct d_bind
  s32 ID;
  s32 y;

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

d_alloc (struct d_alloc *arg):
  ...

arg->ID = g_var;
...

d_bind (struct d_bind *arg):
  if (g_var != arg->ID)
    return -EINVAL;
  /* main functionality */
  ...
```

1. first ioctl
2. copy_to_user
3. second ioctl
4. Check ID with g_var

Example:

```c
Example: System Call Dependencies

fd = open (...
ioctl (fd, D_ALLOC, arg1)
ioctl (fd, D_BIND, arg2)

struct d_alloc
  s32 x;
  s32 ID;

struct d_bind
  s32 ID;
  s32 y;

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

d_alloc (struct d_alloc *arg):
  ...

arg->ID = g_var;
...

d_bind (struct d_bind *arg):
  if (g_var != arg->ID)
    return -EINVAL;
  /* main functionality */
  ...
```

1. first ioctl
2. copy_to_user
3. second ioctl
4. Check ID with g_var
Example: System Call Dependencies

```c
struct d_alloc
s32 x;
s32 ID;

struct d_bind
s32 ID;
s32 y;
```

Example: System Call Dependencies

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

```c
ioctl (fd, cmd, arg):
switch (cmd) {
  case D_ALLOC: d_alloc (arg);
  case D_BIND: d_bind (arg);
  ...
}
```

```c
d_alloc (struct d_alloc *arg):
...
arg->ID = g_var;
...
```

```c
d_bind (struct d_bind *arg):
if (g_var != arg->ID)
  return -EINVAL;
/* main functionality */
...
```

Q. Can be inferred exactly?
Example: System Call Dependencies

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

1. first `ioctl`
2. `d_alloc (struct d_alloc *arg)`:
   ```
   ... 
   arg->ID = g_var;
   ...
   ```
3. second `ioctl`
4. `d_bind (struct d_bind *arg)`:
   ```
   if (g_var != arg->ID)
   return -EINVAL;
   /* main functionality */
   ...
   ```

Q. Can be inferred exactly?
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

```c
ioctl (int fd, unsigned long cmd, void *argp)
write (int fd, void *buf, size_t count)
```
Example: Nested Arguments Structure

```
ioctl (fd, USB_X, arg)
```
Example: Nested Arguments Structure

```c
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

```
struct usbdev_ctrl ctrl;
uchar *tbuf;
...

copy_from_user(&ctrl, arg, sizeof(ctrl))
...
copy_from_user(tbuf, ctrl.data, ctrl.len)

/* do main functionality */
...
```

ioctl (fd, USB_X, arg)

struct usbdev_ctrl:
void *data;
unsigned len;

dst addr

src addr
Example: Nested Arguments Structure

```c
struct usbdev_ctrl ctrl;
uchar *tbuf;

/* do main functionality */
```

```c
copy_from_user(&ctrl, arg, sizeof(ctrl))
```

```c
copy_from_user(tbuf, ctrl.data, ctrl.len)
```

```c
ioctl(fd, USB_X, arg)
```

```c
syscall 9
```
Example: Nested Arguments Structure

```c
struct usbdev_ctrl ctrl;
uchar *tbuf;

... copy_from_user (&ctrl, arg, sizeof(ctrl))
...
... copy_from_user (tbuf, ctrl.data, ctrl.len)
/* do main functionality */
...
```

```c
struct usbdev_ctrl{
    void *data;
    unsigned len;
};
```

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

```c
ioctl_fn__ioctlss[] = {
    ioctl_version,
    ioctl_protover,
    ...
    ioctl_ismountpoint,
};

idx = cmd - INFO_FIRST;
...
funp = _ioctlss[idx];
...
funp(sbi, param);
```
1. Conversion to Direct Control-flow

Before

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

After

```c
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)
```

Compile time conversion: direct control transfer

```
ioctl_fn _ioctls[] = {
    ioctl_version,
    ioctl_protover,
    ...
    ioctl_ismountpoint,
};
```
2. Syscall Dependency Inference

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

```c
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

- static analysis
- <instruction dependency pair>

Linux Kernel

- \( g_{\text{var}} \)
- \( g_{\text{var}} \)

\[
\text{fd} = \text{open}(...) \\
\text{ioctl} (\text{fd}, \text{DALLOC}, \{\text{struct } d\text{-alloc}\}) \\
\text{ioctl} (\text{fd}, \text{DBIND}, \{\text{struct } d\text{-bind}\})
\]
2. Syscall Dependency Inference

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

W: \texttt{g\_var}
R: \texttt{g\_var}

\begin{verbatim}
fd = open (...
ioctl (fd, DALLOC, \{\textbf{struct d\_alloc}\})
ioctl (fd, D_BIND, \{\textbf{struct d\_bind}\})
\end{verbatim}

\textbf{symbolize}
2. Syscall Dependency Inference

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

---

*static analysis*

Linux Kernel

---

*<instruction dependency pair>*

W: g_var
R: g_var

---

symbolize

*write*

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

---

*read*

```
if (g_var == arg->ID) 
...
```

---

*hit*

```
d_alloc (struct d_alloc *arg):
    g_var = gen();
    arg->ID = g_var;
```

---

*hit*

```
d_bind (struct d_bind *arg):
    ...
```

---

*hit*

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

---

*syscalls*

*write*
2. Syscall Dependency Inference

- **Collecting W-R pairs**
- **Runtime validation**
- **Parameter dependency**

Linux Kernel

```
fd = open (...)  
ioctl (fd, D_ALLOC, ...)  
ioctl (fd, D_BIND, ...)  

{struct d_alloc}  
{struct d_bind}
```

```
d_alloc (struct d_alloc *arg):
    arg->ID = g_var;
    g_var = gen();
```

```
d_bind (struct d_bind *arg):
    if (g_var == arg->ID): ...
```

```
write
```

```
read
```

```
symbolize
```
2. Syscall Dependency Inference

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

Linux Kernel

static analysis

W: g_var
R: g_var

<instruction dependency pair>

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

write

syscalls

d_alloc (struct d_alloc *arg):

① address

arg->ID = g_var;
② hit

if yes, true dependency

symbolically tainted

read

① address

d_bind (struct d_bind *arg):

① hit

if (g_var == arg->ID) ...

ioctl (fd, D_ALLOC, {struct d_alloc})
ioctl (fd, D_BIND, {struct d_bind})

write

symbolically tainted

 Terror!
2. Syscall Dependency Inference

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

Linux Kernel

static analysis

W: offset(0x8) R: offset(0x0)

W: g_var R: g_var

<instruction dependency pair>

Collecting W-R pairs

Symbolically tainted

Symbolically tainted

runtime validation

Parameter dependency

if yes, true dependency

W: offset(0x8) R: offset(0x0)

write

d_alloc (struct d_alloc *arg):

W: g_var R: g_var

read

d_bind (struct d_bind *arg):

if (g_var == arg->ID)

fd = open (...)
ioclt (fd, DALLOC)
ioclt (fd, DBIND,

{struct d_alloc}
{struct d_bind})

{struct d_alloc} arg

{struct d_bind} arg

4. Systemcall Dependency Inference

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit

2.2 Address

2.2 Hit
2. Syscall Dependency Inference

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

static analysis

Linux Kernel

W: offset(0x8)
R: offset(0x0)

symbolically tainted

d_bind (struct d_bind *arg):

if (g_var == arg->ID)

fd = open (...)
ioctl (fd, DALLOC, {..._1})
ioctl (fd, DBIND, {..._2})

struct _1 {
    u64 x;
    u32 ID;
}

struct _2 {
    u32 ID,
    u64 x;
}

inferred syscall sequence

prio1: ioctl (fd, D_ALLOC, {..._1})
prio2: ioctl (fd, D_BIND, {..._2})
3. Nested Argument Format Retrieval

```c
struct usbdev_ctrl ctrl; uchar *tbuf;
...
copy_from_user (&ctrl, arg, sizeof(ctrl));
...
copy_from_user (tbuf, ctrl.data, ctrl.len));
...
ioctl (fd, USB_X, arg)
```
3. Nested Argument Format Retrieval

```c
struct usbdev_ctrl ctrl; uchar *tbuf;
...
copy_from_user (&ctrl, arg, sizeof(ctrl));
...
copy_from_user (tbuf, ctrl.data, ctrl.len));
...
ioctl (fd, USB_X, arg)
```

memory view

ctrl:

```
0x8
data
```

0x14

1 hit

syscall
3. Nested Argument Format Retrieval

```c
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

```c
struct usbdev_ctrl ctrl; uchar *tbuf;
...
copy_from_user(&ctrl, arg, sizeof(ctrl));
...
copy_from_user(tbuf, ctrl.data, ctrl.len);
...
ioctl(fd, USB_X, arg)
```

1. **hit**

2. **hit**

symbolically tainted

memory

```c
struct
```
3. Nested Argument Format Retrieval

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

- `ioctl (fd, USB_X, arg)`
- `syscall`
- `hit`
- `memory` tainted
- `symbolically`
- `copy_from_user` (&`ctrl`, `arg`, `sizeof(ctrl)`);
- `symbolic check`
3. Nested Argument Format Retrieval

```c
struct usbdev_ctrl ctrl;
uchar *tbuf;

// Copy from user space
copy_from_user(&ctrl, arg, sizeof(ctrl));

// Copy from user space
copy_from_user(tbuf, ctrl.data, ctrl.len);
```

Diagram:
- **Final Memory View**
  - Upper buffer starting at 0x10
  - Lower buffer starting at 0x8

- **Inferred Syscall Interface**
  - `ioctl(fd, USB_X, {*_1})`
  - `struct _1:`
    - u64 x;
    - {*_2} y;
    - u64 z;
  - `struct _2:`
    - u64 x;
    - u64 y;

- Symbolic check
  - Hit at 0x10
  - Hit at 0x14

Symbolically tainted

- syscall
- ioctl
- Inferred syscall interface

Inferred syscall interface structure:
- struct _1:
  - u64 x;
  - {*_2} y;
  - u64 z;
- struct _2:
  - u64 x;
  - u64 y;

Final memory view:
- Upper buffer starting at 0x10
- Lower buffer starting at 0x8

Implementation

1. **Syzkaller**
   - send unsolved conds
   - process solved conditions

2. **S2E**
   - constraint solving
   - symbolic checking

3. **GCC**
   - convert to direct control-flow

4. **SVF/LLVMLINUX**
   - collect dependency set

5. **Python-based**
   - transfer data

- calling orders
- argument retrieval
- infer
- unsolved conds
- solved
- ondemand exec
- convert
- collect dependency set
- transfer data
- send unsolved conds
- process solved conditions

- candidate dependency pairs
- feedback
- static analysis
- hybrid-fuzzing
- Linux Kernel
- *Linux Kernel
- SVF/LLVMLINUX
- GCC
- Syzkaller
- Python-based
- S2E
- Linux Kernel
- Agent
- Symbolic Analyzer
- unsolved conds
- solved
- convert
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

HFL-hybrid
HFL-direct
HFL-struct
HFL-full
baseline

Coverage (%)

ext4
ppp
rds
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