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, †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 OSs.

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

Q. Can be fuzzed enough to explore all functions?

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

\text{ioctl_fn} \_ _\text{ioctls}[] = \{ \\
\text{ioctl_version, ioctl_protover}, \\
\ldots \\
\text{ioctl_ismountpoint}, \\
\};

<indirect function call>

indirect control transfer

derived from syscall arguments

targets to be hit
Challenge 2: System Call Dependencies

- **explicit syscall dependencies**
  \[
  \begin{align*}
  \text{int} & \quad \text{open} \ (\text{const char *pathname, int flags, mode_t mode}) \\
  \text{ssize_t} & \quad \text{write} \ (\text{int fd, void *buf, size_t count})
  \end{align*}
  \]

- \[
  \begin{align*}
  \text{ioctl} & \ (\text{int fd, unsigned long req, void *argp}) \\
  \text{ioctl} & \ (\text{int fd, unsigned long req, void *argp})
  \end{align*}
  \]

Q. What dependency behind?
Example: System Call Dependencies

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

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 */
  ...

Q. Can be inferred exactly?

Copy to user:

Read

Write

1. first ioctl
2. d_alloc (structure d_alloc *arg)
3. second ioctl
4. d_bind (structure d_bind *arg)

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)`
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 */
...

/* do main functionality */
```
HFL: Hybrid Fuzzing on the Linux Kernel

- **Handling the challenges**
  - calling orders
  - argument retrieval

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

- **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
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`,

```c
};
```
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)

symbolically tainted

write

<instruction dependency pair>

struct 1 { u64 x; u32 ID; }

struct 2 { u32 ID, u64 x; }

prio1: ioctl(fd, DALLOC, {*_1})
prio2: ioctl(fd, D_BIND, {*_2})

inferred syscall sequence

if (g_var = arg->ID)
...
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);
…
```

**Final memory view**
- Upper buffer: 0x10, 0x8
- Lower buffer: 0x10

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

Symbolically tainted

1. Hit
2. Hit
Implementation

1. **Syzkaller**
   - send unsolved conditions
   - 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

Diagram:
- Fuzzer
- Agent
- Symbolic Analyzer
- Linux Kernel
- *Linux Kernel

- unsolved conditions
- solved
- ondemand exec
- convert
- hybrid-fuzzing
- candidate dependency pairs
- feedback
- calling orders
- argument retrieval
- infer

Feedback process:
- Collect unsolved conditions from Fuzzer
- Convert to direct control-flow with GCC
- Collect dependency set with SVF
- Transfer data with Python-based approach
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
Case Study: Syscall Dependency

Handled by hybrid feature

1st ioctl

Handled by hybrid feature

2nd ioctl

a var

ID written to arg

Read ID from arg

check ID

FAIL!!

SUCCESS!!

Covered by syscall dependency inference

prio1: ioctl(fd, PPPNEWUNIT, ID)
prio2: ioctl(fd, PPPCONNECT, ID)

Covered by syscall dependency inference

1

2

1st ioctl

2nd ioctl

ID written to arg

Read ID from arg

check ID

SUCCESS!!

FAIL!!

Covered by syscall dependency inference

prio1: ioctl(fd, PPPNEWUNIT, ID)
prio2: ioctl(fd, PPPCONNECT, ID)

Covered by syscall dependency inference

1st ioctl

2nd ioctl

ID written to arg

Read ID from arg

check ID

FAIL!!

SUCCESS!!

Covered by syscall dependency inference

prio1: ioctl(fd, PPPNEWUNIT, ID)
prio2: ioctl(fd, PPPCONNECT, ID)

Covered by syscall dependency inference
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