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

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?

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

targets to be hit

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

<indirect function call>

indirect control transfer

derived from syscall arguments

<function pointer table>

5
Challenge 2: System Call Dependencies

explicit syscall dependencies

\[
\begin{align*}
\text{int open} & \quad (\text{const char } *\text{pathname}, \text{int flags, mode}_t \text{ mode}) \\
\text{ssize}_t \text{ write} & \quad (\text{int fd, void } *\text{buf, size}_t \text{ count}) \\
\end{align*}
\]

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

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?

Check ID with g_var

Copy to User

First ioctl

Second ioctl

Read

Write

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

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

```
void *data;
unsigned len;
```

Q. Can be inferred exactly?
HFL: Hybrid Fuzzing on the Linux Kernel

- The first hybrid kernel fuzzer
- Handling the challenges

1. Implicit control transfer
   - Convert to direct control-flow
2. System call dependencies
3. Complex argument structure
   - Infer nested argument structure
   - Agent act as a glue between the two components

1. Coverage-guided/system call fuzzer
2. Hybrid fuzzing

- Candidate dependency pairs
- Static analysis
- Calling orders
- Argument retrieval
- Unsolved conds
- Convert
- Solve
- Ondemand exec
- Inputs
- Feedback

- Symbolic Analyzer
- Fuzzer
- Agent

hybrid-fuzzing
1. Conversion to Direct Control-flow

Before:

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

After:

```c
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

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

1. Collecting W-R pairs
   - $fd = \text{open}(\ldots)$
   - $\text{ioctl}(fd, D\_ALLOC, \{\_1\})$
   - $\text{ioctl}(fd, D\_BIND, \{\_2\})$

2. Runtime validation
   - $\text{write}(\ldots)$
   - $\text{read}(\ldots)$

3. Parameter dependency
   - $\text{d\_bind}(\text{struct d\_bind } *arg)$:
     - if($g\_var == \text{arg}\rightarrow ID$)

---

Symbolically tainted

Windows dependency pair:

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

Symbolically tainted

Linux Kernel

Struct $\_1$
- u64 x;
- u32 ID;

Struct $\_2$
- u32 ID,
- u64 x;

Inferred syscall sequence:

$prio1: \text{ioctl}(fd, D\_ALLOC, \{\_1\})$
$prio2: \text{ioctl}(fd, D\_BIND, \{\_2\})$
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);

…
```

---

**final memory view**

```
struct upper buffer
    0x10
struct lower buffer
    0x8
```

**inferred syscall interface**

```
ioclt(fd, USB_X, &ctrl);  

struct _1:  
    u64 x;  
    u64 y;  
    u64 z;
```

---

Symbolically tainted

```
struct _2:  
    u64 x;  
    u64 y;  
```

---

**symbolic check**
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

**Fuzzer**
- inputs
- convert
- candidate dependency pairs
- unsolved conds
- solved
- feedback
- calling orders
- argument retrieval
- infer
- ondemand exec

**Agent**
- solved
- process solved conditions

**Symbolic Analyzer**
- unsolved conds
- convert
- *Linux Kernel
- hybrid-fuzzing

**Linux Kernel**
- convert
- *Linux Kernel
- collect dependency set

**Linux**
- convert to direct control-flow

**Agent**
- infer
- calling orders
- argument retrieval

**Symbolic Analyzer**
- unsolved conds
- convert
- *Linux Kernel
- hybrid-fuzzing
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
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