HEALER: Relation Learning Guided Kernel Fuzzing

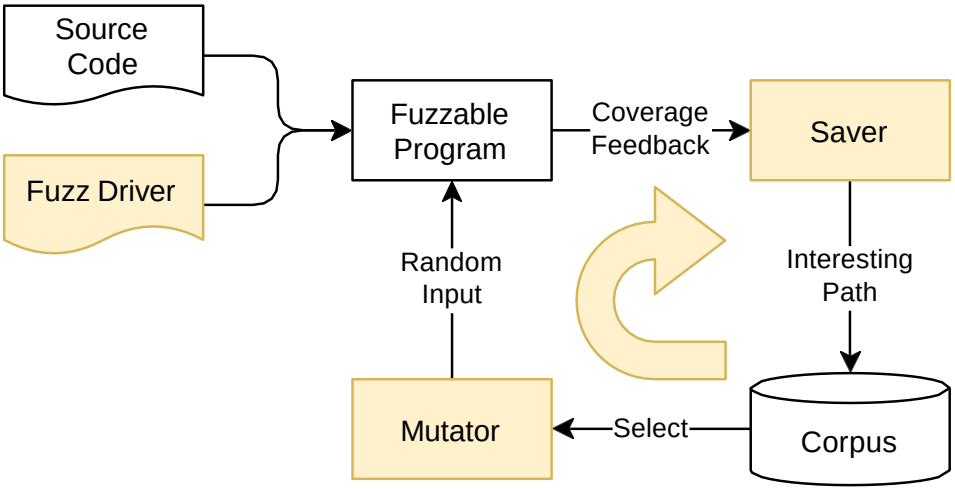
Hao Sun[1], Yuheng Shen[1], Cong Wang[1], Jianzhong Liu[1], Yu Jiang✉[1], Ting Chen✉[2], Aiguo Cui[3*]
KLISS, BNRist, School of Software, Tsinghua University, Beijing, China[1]
Center for Cybersecurity, University of Electronic Science and Technology of China, Chengdu, China[2]
Huawei Technologies Co., Ltd, China[3]

Outline

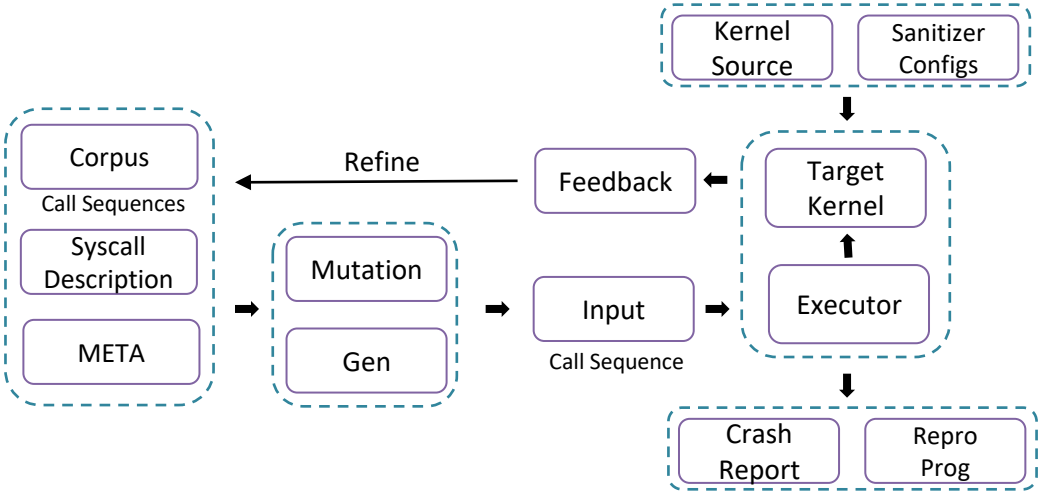
1. Coverage Guided Kernel Fuzzing
2. Motivation
3. Relation Learning
4. Implementation
5. Evaluation
6. Future Work

Coverage-Guided Kernel Fuzzing

Fuzzing 101


Source Code
Fuzz Driver
Fuzzable Program
Coverage Feedback
Saver
Random Input
Interesting Path
Mutator
Select
Corpus

Coverage Guided Kernel Fuzzing



Coverage Guided Kernel Fuzzing: Input

Input

Call Sequence

```
sock_fd = socket(AF_INET, SOCK_STREAM, 0)
bind(sock_fd, &addr, sizeof(addr)
listen(sock_fd, …)
accept(sock_fd, &peer_addr, &size)
```

- Structure and partial semantic aware
- Setup kernel state
- Reach deep kernel logic

Syscall Description Language: Syzlang

```
resource sock[fd]
resource sock_in[sock]

socket$inet(domain const[AF_INET], type flags[socket_type], proto int32) sock_in
accept$inet(fd sock_in, peer ptr[out, sockaddr_in], peerlen ptr[inout, len[peer, int32]]) sock_in
bind$inet(fd sock_in, addr ptr[in, sockaddr_in], addrlen len[addr])
listen(fd sock, backlog int32)
```

- Rich type, type constructors
- Semantic modifier
- Encoding accurate structure, partial semantics

Syscall Description Language: Syzlang

```
resource sock[fd]
resource sock_in[sock]

socket$inet(domain const[AF_INET], type flags[socket_type], proto int32) sock_in
accept$inet(fd sock_in, peer ptr[out, sockaddr_in], peerlen ptr[inout, len[peer, int32]]) sock_in
bind$inet(fd sock_in, addr ptr[in, sockaddr_in], addrlen len[addr])
listen(fd sock, backlog int32)
```

- Resource Type
  - Value, output from other calls
  - `resource A[B]` => A is subtype of B
- Call Specialization
  - Specialize partial arguments
  - Format: call$name

Motivation

Call Combination

For a set of syscalls $\{ S_0, S_1, S_2, \ldots S_n \}$

Generate a sequence of calls $[C_0, C_1, C_2, \ldots]$

For sequence $[C_0, C_1]$, how to choose the next system call?

Random? Optimized strategy?

What is the idea behind the strategy ?

Problem: Countless Call Combinations

- Around 400 syscalls in Linux
- 4000+ specialized syscalls in Syzlang description
- Length of generated call sequence is 8~32
- Possible number of combinations is $\sum_{k=8}^{32}\binom{4000}{k} \approx 10^{80}$
- $10^2 \sim 10^3$ exec/s, 10^{69} years for all combinations
- Most call sequences are invalid, equivalent

Need better strategy to choose call combination, rather than random.

Choice Table Of Syzkaller

- Each item records the probability that a syscall should be invoked before another syscall
- Example
  - For sequence [C_0, C_1], random choose C_i
  - Choose next call based on the probability

$$P_{ij} = \frac{(p0_{ij} * p1_{ij})}{1000}$$

- Calculated by an empirical analysis algorithm
- Static part $p0_{ij}$
  - Hard-coded value for each type
  - Sum of common type
- Dynamic part $p1_{ij}$
  - Count each adjacent calls
  - ulate sum

Too Empirical

Observation: Influence Relation

```
sock_fd = socket(AF_INET, SOCK_STREAM, 0)
```

Create socket inside the kernel

```
bind(sock_fd, &addr, sizeof(addr))
```

Bind address to the socket

```
listen(sock_fd, …)
```

Mark socket

```
accept(sock_fd, &peer_addr, &size)
```

- Former calls setup related kernel states
- The latter calls can be influenced by those states
- Execution path of the latter call changed due to the internal kernel state modified by the former call

Influence relations exist between two system calls if the execution of a former can alter the latter's execution path.

Observation: Guide with Influence Relation

```
sock_fd = socket(AF_INET, SOCK_STREAM, 0)
```

Create socket inside the kernel

```
bind(sock_fd, &addr, sizeof(addr))
```

Bind address to the socket

```
listen(sock_fd, …)
```

Mark socket

```
accept(sock_fd, &peer_addr, &size)
```

- Influence relation exists between calls
- Some execution paths of one call may only be executed in certain kernel states
- Insert more system calls that have influence relations before the target system call so that we can trigger different kernel states and allow each system call to enter deep execution paths

The number of invalid test cases and the size of the search space can be reduced significantly by taking relations between system calls into consideration.

Our Idea

Guide Kernel Fuzzing with Relation Learning

Learn the influence relations dynamically, iteratively

Guide generation and mutation with learned relations

Increase the quality of inputs, speedup the fuzzing process

HEALER: Relation Learning

Definition

Relation

A system call C_i has an influence on another system call C_j if the execution of C_i can influence the execution path of C_j by modifying the kernel's internal state.

- Relation is about influence of execution path
- The reason behind relation is kernel state

Static Learning

- **Purpose**: Learn the relations expressible by Syzlang description.
- **Idea**: The producer syscall of one resource can influence the consumer syscall of that resource.
- **Steps**: two simple rules:
  - The **return type** of C_i is a resource type $r0$, or any parameter in C_i is a **pointer** of this resource type with an **outward** data flow direction
  - At least one of C_j's parameters is a resource type $\boldsymbol{r0}$ or resource type $\boldsymbol{r1}$ that is compatible with $r0$ with an inward data flow direction

Static Learning

```
socket$inet(domain const[AF_INET], type flags[socket_type], proto int32) sock_in
bind$inet(fd sock_in, addr ptr[in, sockaddr_in], addrlen len[addr])

socketpair(domain flags[domain], type flags[socket_type], proto int32, fds ptr[out, sock_pair])
bind$inet(fd sock_in, addr ptr[in, sockaddr_in], addrlen len[addr])

resource sock_in[sock]
socket$inet(domain const[AF_INET], type flags[socket_type], proto int32) sock_in
listen(fd sock, backlog int32)
```

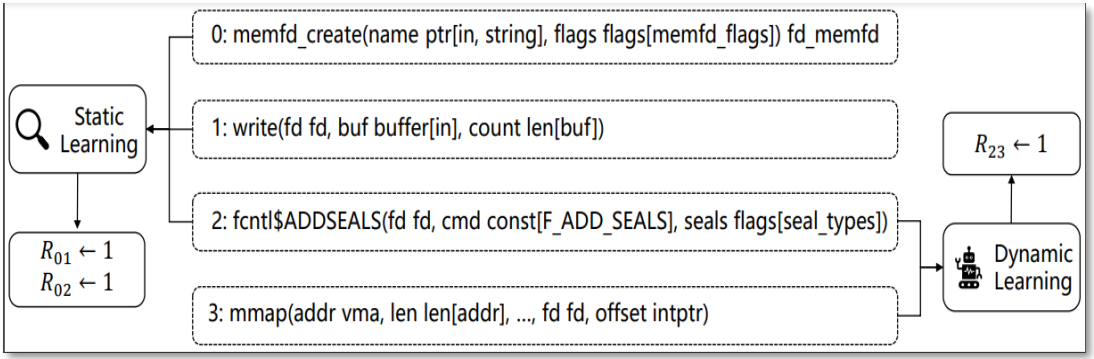
Dynamic Learning: Minimization

- **Purpose**: Only analyze calls that contribute to new coverage.
- **Idea**: Remove one call as long as the coverage keeps the same.
- **Steps**: For sequence P and its coverage:
  - Extract the subsequence p' for call Ci that has not yet been included in the other minimal sequences
  - Remove each call C' before Ci in p'
  - If coverage keeps same, commit the change
- **Example**
  - For [memfd_create, write, fcntl, mmap], with [cov0, cov1, cov2, cov3], cov3 contains new coverage
  - `write` is removed, the final sequence is [memfd_create, fcntl, mmap]

Dynamic Learning

- **Purpose**: Learn the relations not expressible by Syzlang description
- **Idea**: The relation is all about execution path, observe the coverage change
- **Steps**: For each **adjacent call pair** (C_i, C_j) of the minimized sequence P:
  - Remove C_i, observe the coverage change of C_j
  - If the coverage of C_j changed, C_i must have influence relation with C_j , since they're adjacent

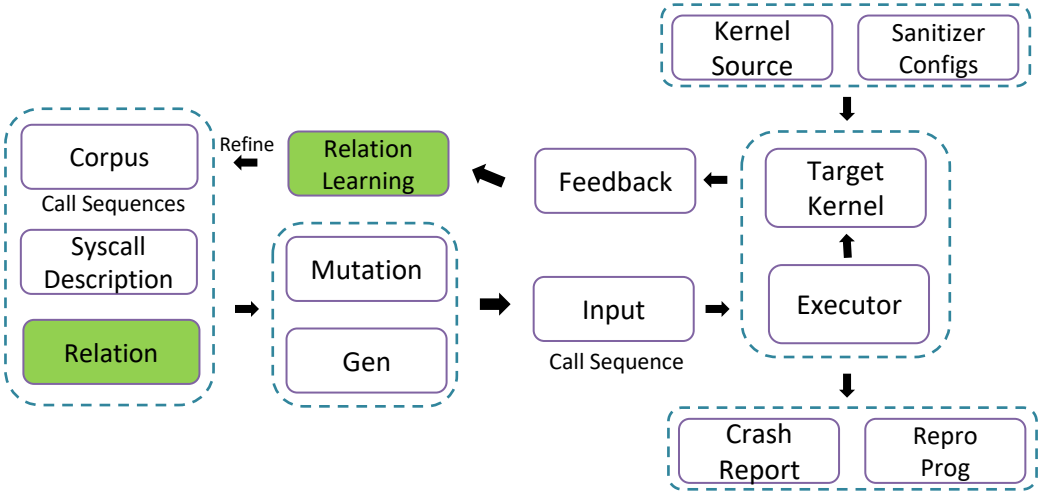
Dynamic Learning



Guided Generation and Mutation

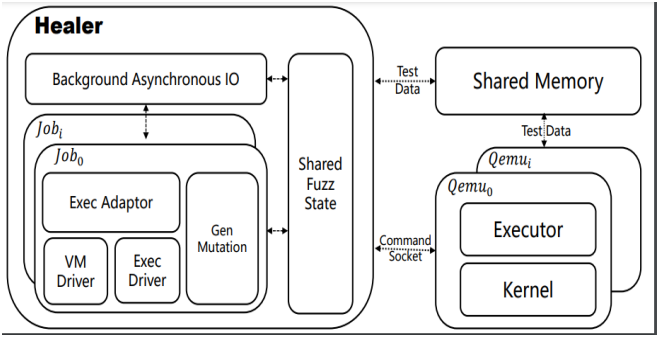
- **Purpose**: Generate high quality inputs
- **Idea**: Use learned relations to select call that matters
- **Steps**: For call sequence [C_i , C_j , C_k]:
  - Find candidate calls that can be influenced by C_i , C_j , C_k, count number of calls that influence the candidate as weight.
  - Choose weighted
- **Example:**
  - For sequence [socket, bind]
  - The candidates are **[listen: 2, accept: 1]**
  - `**listen**` has higher priority to be chosen

Revisit the Fuzzing Loop



Implementation

Our arch: Keep It Simple



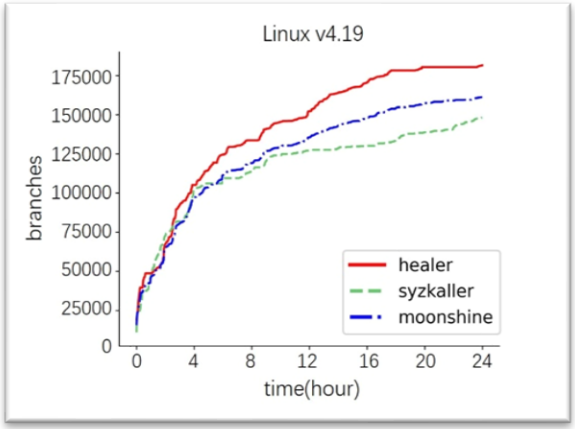
- Shared fuzzer states
  - Fuzzer runs in host
  - Only executor runs in VM
- Shmem based communication
  - QEMU ivshm
- Modular Design

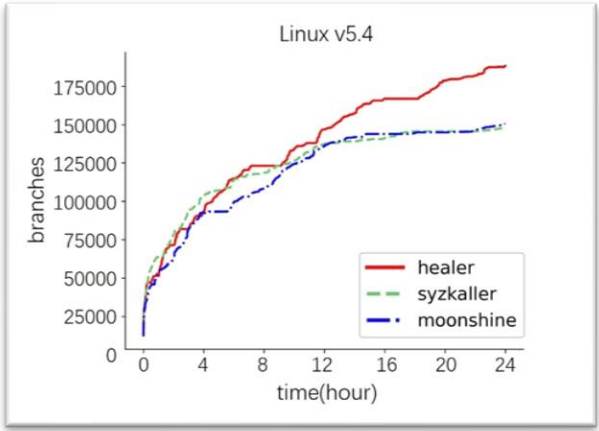
Implementation

- Implement from scratch
- Written in rust, 16064 loc
- Store relations in high performance `HashMap` (`Ahash`)
- Leverage `tokio` to implement background IO
- Read-Write lock & atomic operation, reduce sync overhead

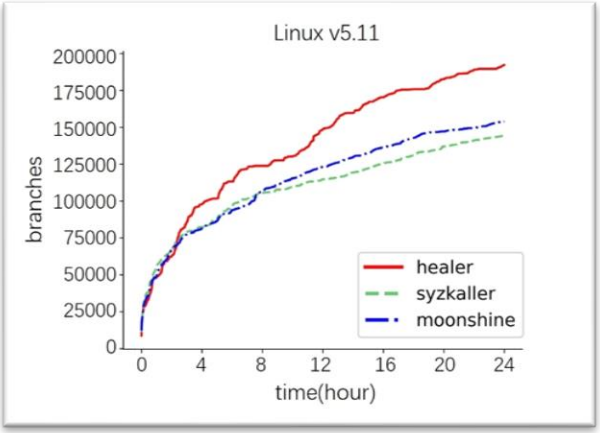
Evaluation

Improvement





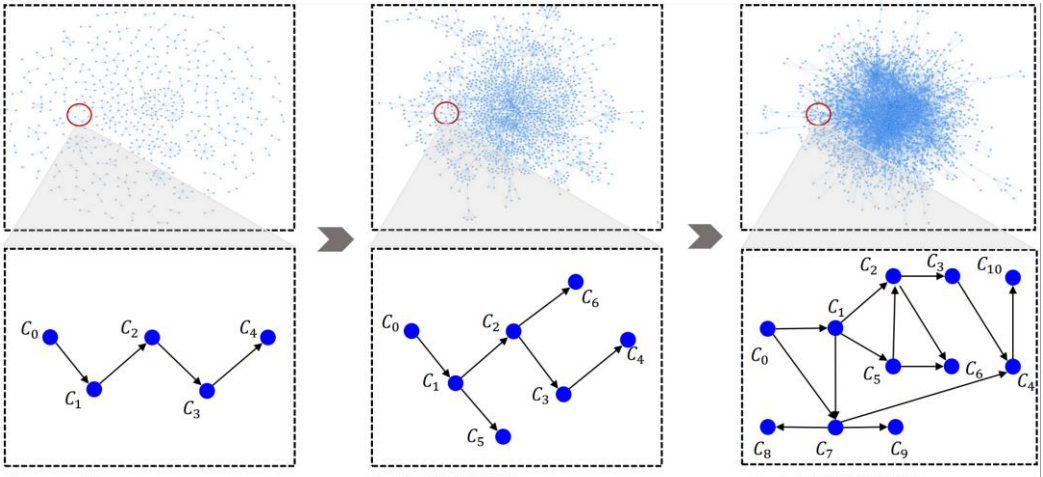
Improvement



- Syzkaller: +28%, +2.2x
- Moonshine: +21%, +1.8x
- Syzkaller: 18+ bugs
- Moonshine: 15+ bugs

The coverage improvement, bug detection improvement are obvious.

Learned Relations



C_0: openat$kvm
C_1: ioctl$KVM_CREATE_VM
C_2: ioctl$KVM_CREATE_VCPU
C_3: ioctl$KVM_SET_USER_MEMPRY_REGION

C_4: ioctl$KVM_RUN
C_5: ioctl$KVM_CREATE_IRQCHIP
C_6: ioctl$KVM_ENABLE_CAP_CPU
C_7: ioctl$KVM_SET_LAPIC

C_8: ioctl$KVM_IRQ_LINE
C_9: ioctl$KVM_SMI
C_{10}: ioctl$KVM_SET_GUEST_DEBUG

Long Time Fuzzing

- Experiments
  - Fuzzing for 2 weeks, multiple versions, e.g., 4.19, 5.4, 5.6.
  - Found 218 bugs in total
  - 33 are previously-unknown
- In practice
  - Long time fuzzing in internal server
  - Report 20+ bug/week
  - 3~6 confirm/week

Long Time Fuzzing

> no fault-injection log was printed before the task hang.

OK, then it seems like a big problem.

Any workload log from the fuzzer so we

Or just using the tool?

Tha

Qu

>

> If you fix this issue, please add the following tag to the commit:
> Reported-by: Hao Sun <sunhao.th@gmail.com>

Thanks for reporting it.

•••

Given wait_event() in synchronize_rcu_expedited(), it is no good to come in a rcu context.

Matthew Wilcox <willy@infradead.org>
发送至 我、 akpm、 linux-mm、 linux-kernel

英语 > 中文 翻译邮件

>> If you fix this issue, please add the following tag to the commit:
>> Reported-by: Hao Sun <sunhao.th@gmail.com>
>
> This is probably a dup, causes skb_expand_head() changes,
> CC Vasily Averin <vvs@virtuozzo.com> is currently working on a fix.

Thank you for this report and especially for C reproducer!
Vasily Averin

•••

ONLY_THP_FOR_FS.

allocated by khugepaged.
re's no knowledge of THPs in

e. Somebody needs to figure out
split -- was this ioctl issued through

rmation.

Future Work

- Integrate to upstream (CI)
- Implement `hub` to support fuzzing on multiple hosts
- Reduce the manual efforts of writing syscall description

Thank You