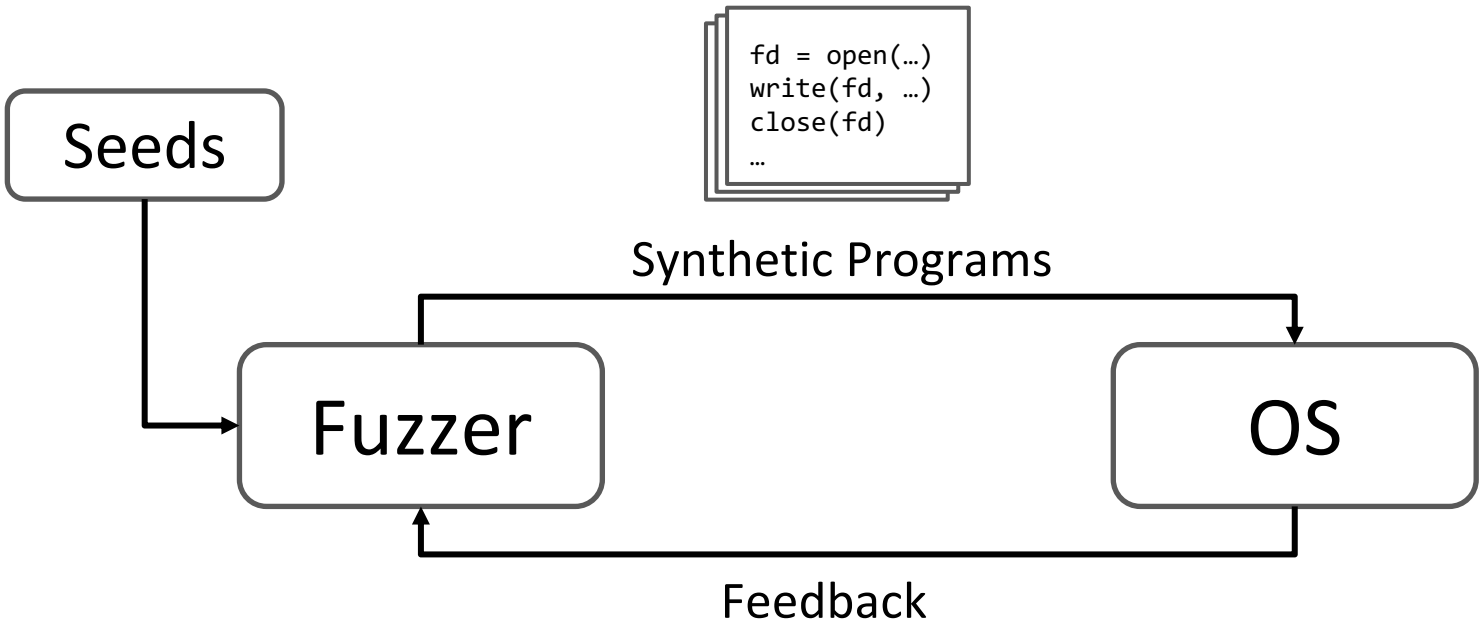
MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, Suman Jana
Columbia University

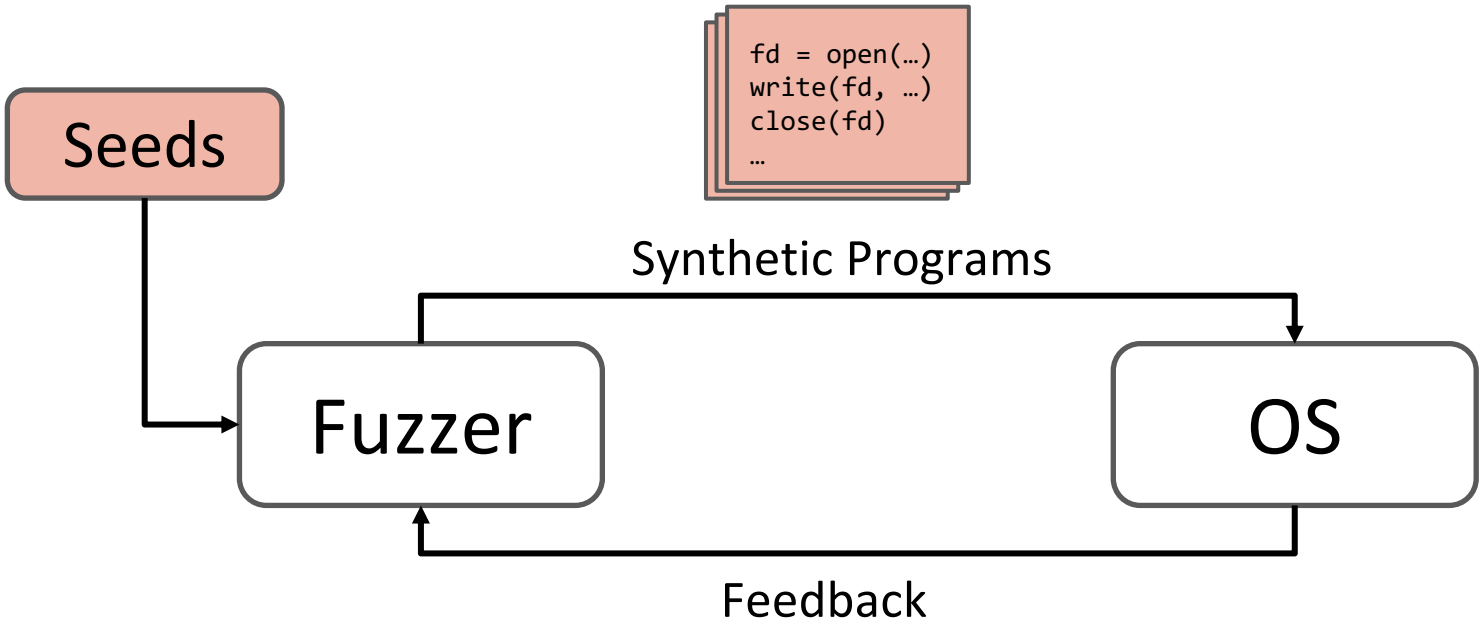
OS Fuzzing

- Popular technique to find OS vulnerabilities
- Primarily tests system-call interface
  - Can be invoked by untrusted user programs
  - Large surface area for attack

OS Fuzzing - Overview

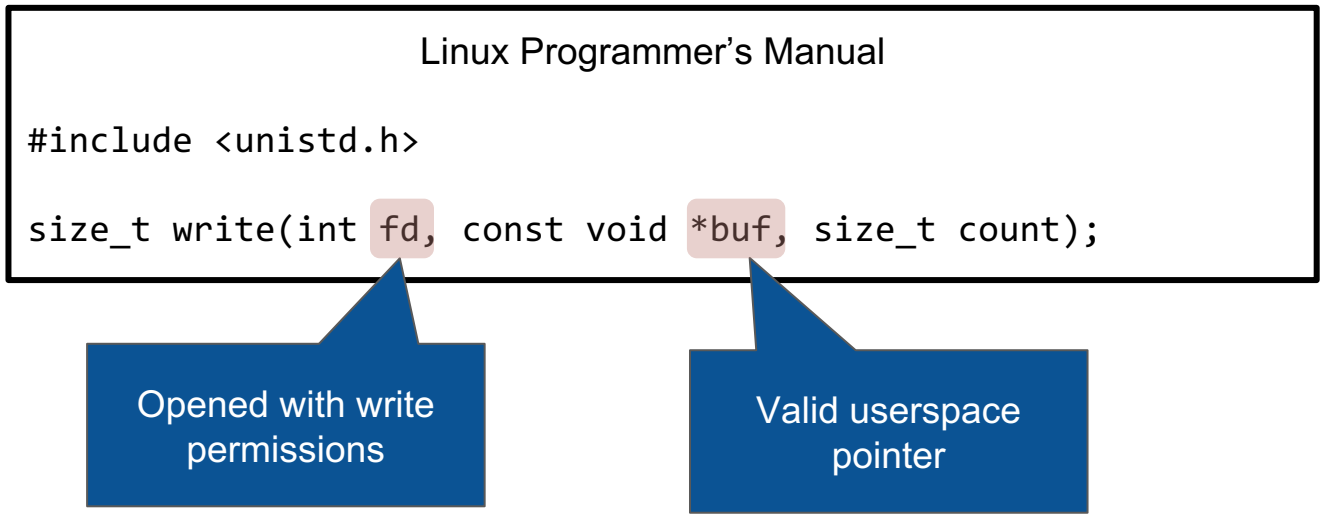


OS Fuzzing - Overview



Synthetic Program Generation

- Goal – Maximize code coverage
- Random generation alone is unlikely to succeed



Synthetic Program Generation

- Goal – Maximize code coverage
- Random generation alone is unlikely to succeed
  - Fuzzer must track and maintain system-call dependencies

Synthetic Program Generation

- Goal – Maximize code coverage
- Random generation alone is unlikely to succeed
  - Fuzzer must track and maintain system-call dependencies
- State-of-the-art – Thousands of hardcoded rules!!

```
resource fd[int32]
...
open(file ptr[in], …) fd
write(f fd, buf buffer[in], count len[buf])
```

Synthetic Program Generation

- Goal – Maximize code coverage
- Random generation alone is unlikely to succeed
  - Fuzzer must track and maintain system-call dependencies
- State-of-the-art – Templates with *thousands* of manual rules
- Hard to scale

MoonShine

Real Program Traces

Distilled Seeds

Trace Distillation vs. User-Level Seed Selection

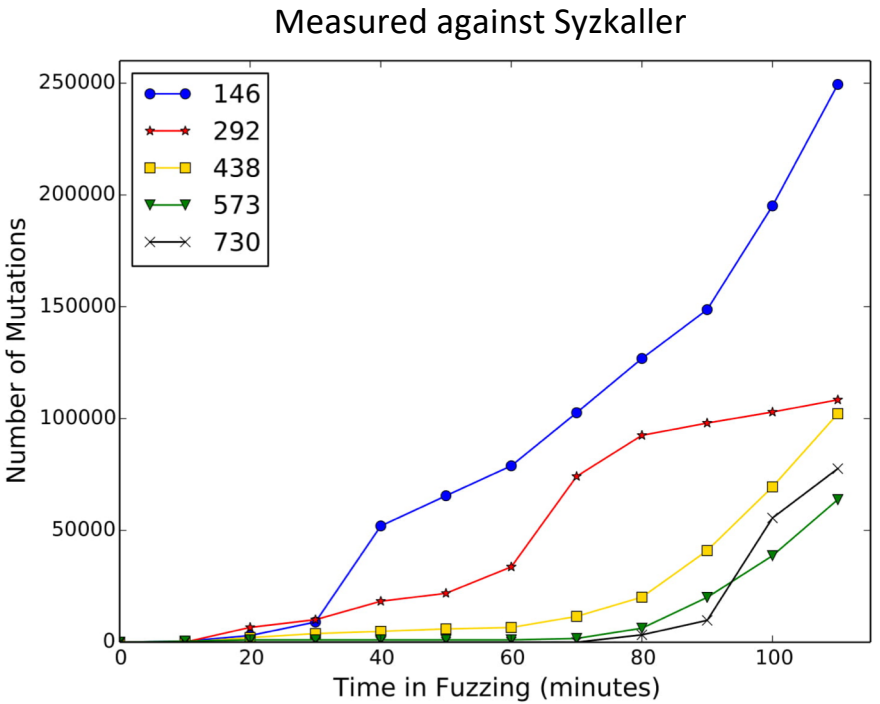
- MinSet (Sec'14)
  - Find smallest subset with most coverage

Distillation Challenges

- Minimize trace sizes
- Track dependencies

Why does trace size matter?

- Fuzzer performance tied to program size (# calls)
- 10 second trace of Chromium contains 462,225 calls!!
- Traces can't be directly used as seeds



Trace Distillation

- Goal – Minimize the traces while preserving coverage

- Strategy – Select calls that contribute most coverage

Dependencies

- Explicit Dependencies
  - Shared state passed through arguments
- Implicit Dependencies
  - Modify shared kernel data structure

Explicit Dependencies

- Call **A** is *explicitly dependent* on call **B** if **B** produces a result used by **A**

```
3 = open(“/tmp/file0.txt”, O_WRONLY)
16 = write(3, “somerandomtext\n”, 16)
```

Implicit Dependencies

- Call **A** is *implicitly dependent* on Call **B** if **B** affects the execution of **A** by modifying a shared kernel data structure

mlockall(int lock_flags)

```
int mlockall(…) {
…
void mlock_fixup_lock {
    …
    if (lock)
        vma->vm_flags = lock_flags
}
```

msync(void *addr, size_t length, int flags)

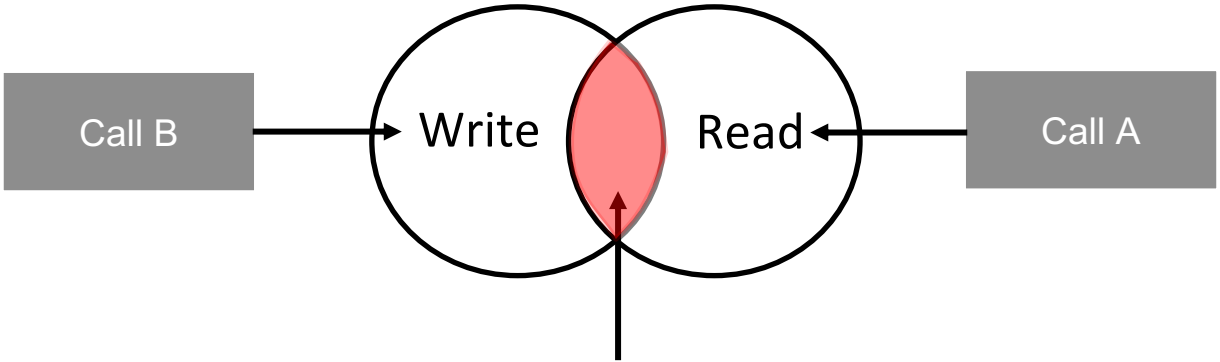
```
int msync(…) {
    …
    if (vma->vm_flags & VM_LOCKED)
        error = -EBUSY
    …
}
```

Tracking Explicit Dependencies

- Statically analyze trace
- Return Cache: Map<(Type, Ret-Val), List<Call>>
  - *data* type or *semantic* type (e.g., file descriptor)
- If (type, value) key in Return Cache, then every call that returned this key is marked as explicit dependency.

Tracking Implicit Dependencies

- Control and Data Flow Analysis
- Call **c** uses shared variable **v** in conditional ⇒ **c** is *read dependent on* **v**
- Call **c** writes to shared variable **v** ⇒ **c** is *write dependent on* **v**



Working Example

Trace Excerpt

```
mlockall(MCL_FUTURE)
3 = open(…)
0x7b2000 = mmap(NULL, …, 3, 0)
0x7b3000 = mmap(NULL, …, 3, 0)
0x7b4000 = mmap(NULL, …, 3, 0)
-EBUSY = msync(0x7b2000, …,
               MS_INVALIDATE)
5 = write(1, “Hello”, 5)
3 = write(1, “abc”, 3)
```

Distilled Trace

Working Example

Trace Excerpt

```
mlockall(MCL_FUTURE)
3 = open(…)
0x7b2000 = mmap(NULL, …, 3, 0)
0x7b3000 = mmap(NULL, …, 3, 0)
0x7b4000 = mmap(NULL, …, 3, 0)
-EBUSY = msync(0x7b2000, …,
               MS_INVALIDATE)
5 = write(1, “Hello”, 5)
3 = write(1, “abc”, 3)
```

Distilled Trace

Working Example

Trace Excerpt

```
mlockall(MCL_FUTURE)
3 = open(…)
0x7b2000 = mmap(NULL, …, 3, 0)
0x7b3000 = mmap(NULL, …, 3, 0)
0x7b4000 = mmap(NULL, …, 3, 0)
-EBUSY = msync(0x7b2000, …,
               MS_INVALIDATE)
5 = write(1, “Hello”, 5)
3 = write(1, “abc”, 3)
```

Distilled Trace

Working Example – Explicit Dependencies

Trace Excerpt

```
mlockall(MCL_FUTURE)
3 = open(…)

0x7b3000 = mmap(NULL, …, 3, 0)
0x7b4000 = mmap(NULL, …, 3, 0)



3 = write(1, “abc”, 3)
```

Distilled Trace

```
0x7b2000 = mmap(NULL, …, 3, 0)
-EBUSY = msync(0x7b2000, …,
                MS_INVALIDATE)
5 = write(1, “Hello”, 5)
```

Explicit Dependencies

Working Example – Implicit Dependencies

Trace Excerpt

```
mlockall(MCL_FUTURE)

0x7b3000 = mmap(NULL, …, 3, 0)
0x7b4000 = mmap(NULL, …, 3, 0)

3 = write(1, “abc”, 3)
```

Distilled Trace

```
3 = open(…)
0x7b2000 = mmap(NULL, …, 3, 0)
-EBUSY = msync(0x7b2000, …,
               MS_INVALIDATE)
5 = write(1, “Hello”, 5)
```

Implicit Dependencies

Implementation

- Linux Kernel
- Syzkaller – OS Fuzzer (Google)
- Strace – System-call traces
- Kcov – Coverage
- Smatch – Static analysis framework
  - Read deps. with Condition Hook
  - Write deps. with Unary Op and Assign. Hooks
- 2580 lines of Golang and 640 lines of C

Evaluation - Setup

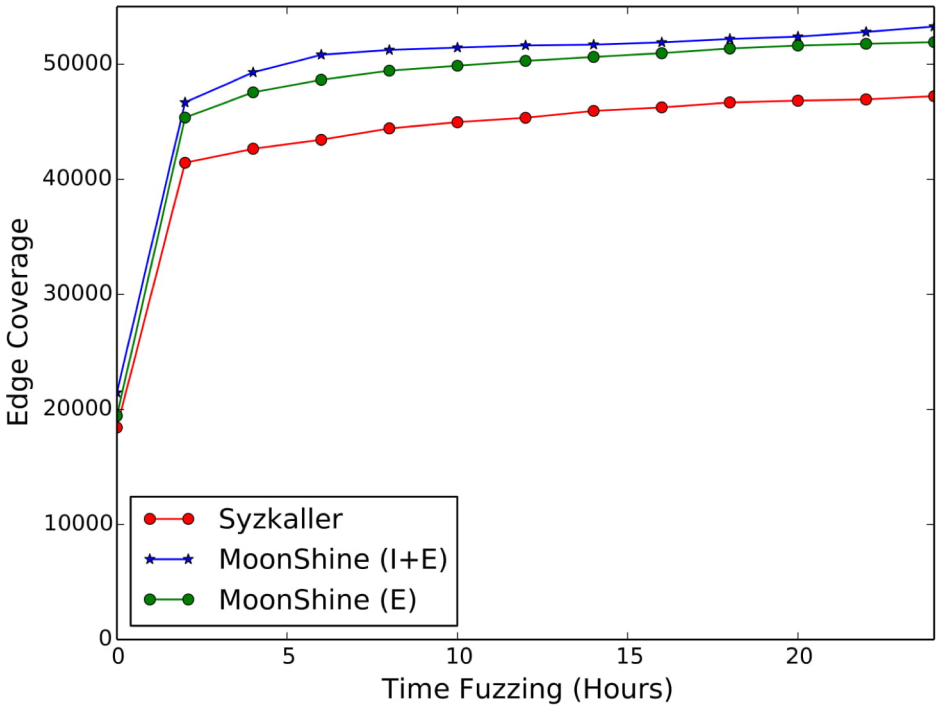
Seed Source	Number of Traces
Glibc Testsuite	1120
Linux Kernel Selftests	55
Linux Testing Project (LTP)	390
Open Posix Testsuite	1630

New Vulnerabilities

Subsystem	Module	Operation	Impact	Version Introduced	Distill. Method
BPF	bpf/devmap.c	dev_map_alloc()	Illegal allocation size	4.0	(I+E) & (E)
BTRFS	fs/btrfs/file.c	btrfs_fallocate()	Assert Failure	4.14	(I+E)
Ext4	fs/fs-writeback.c	move_expired_inodes()	Use After Free	4.6	(I+E)
JFS	fs/jfs/xattr.c	__jfs_setxattr()	Memory Corruption	2.6	(I+E) & (E)
Network	net/ipv4/inet_connection_sock.c	inet_child_forget()	Use after Free	4.4	(I+E)
Network	net/core/stream.c	sk_kill_stream_queues()	Memory Corruption	4.4	(I+E)
Network	net/core/dst.c	dst_release()	NULL Pointer Deref	4.15-rc8	(I+E)
Network	net/netfilter/nf_conntrack_core.c	init_conntrack()	Memory Leak	4.6	(I+E)
Network	net/nfc/nfc.h	nfc_device_iter_exit()	NULL Pointer Deref	4.17-rc4	(I+E)
Network	net/socket.c	socket_setattr()	NULL Pointer Deref	4.10	(I+E) & (E)
Posix-timers	kernel/time/posix-cpu-timers.c	posix_cpu_timer_set()	Integer Overflow	4.4	(I+E) & (E)
Reiserfs	fs/reiserfs/inode.c, fs/reiserfs/ioctl.c, fs/direct-io.c	Multiple	Deadlock	4.10	(I+E)
TTY	tty/serial/8250/8250_port.c	serial8250_console_putchar()	Kernel Hangs Indefinitely	4.14-rc4	(I+E)
VFS	fs/iomap.c	iomap_dio_rw()	Data Corruption	3.10	(I+E) & (E)
VFS	lib/iov_iter.c	iov_iter_pipe()	Data Corruption	3.10	(I+E) & (E)
VFS	fs/pipe.c	pipe_set_size()	Integer Overflow	4.9	(I+E) & (E)
VFS	inotify_fsnotify.c	inotify_handle_event()	Memory Corruption	3.14	(I+E)

Coverage Improvement

- 13.1% coverage increase over default Syzkaller with implicit + explicit
- 9.7% coverage increase over default Syzkaller with only explicit



Effectiveness of Distillation

Total Calls	After Distillation	Comparison	Coverage Preserved
2,900,000	16,400	176x reduction	86%

Vulnerability Discovered By MoonShine

Exhibit: Buffer Overflow in inotify (CVE-2017-7533)

CPU 1

```
inotify_handle_event(..., file_name)
{
  //file_name is currently HelloWorld
  len = strlen(file_name);
  alloc_len += len + 1;
  event = kmalloc(alloc_len, GFP_KERNEL);
```

CPU 2

```
sys_rename(..., new_name)
{
  //new_name is LongFileName
   …
   copy_name(file_name, new_name)
  //file_name changed to LongFileName
}
```

Privilege Escalation

```
   strcpy(event->name, file_name);
   //strcpy will now overflow event
}
```

Exhibit: Buffer Overflow in inotify (CVE-2017-7533)

Seed Distilled by MoonShine

```
1:  mmap(...)
2:  r0 = inotify_init ()
3:  r1 = inotify_add_watch(r0, ".", 0xfff)
4:  chmod(".",  0x1ed)
5:  r2 = creat("short1" , 0x1ed)
6:  close(r2)
7:  rename("short1" , "short2" )
8:  close(r0)
```

Exhibit: Buffer Overflow in inotify (CVE-2017-7533)

Seed Distilled by MoonShine

```
1:  mmap(...)
2:  r0 = inotify_init ()
3:  r1 = inotify_add_watch(r0, ".", 0xfff)
4:  chmod(".",  0x1ed)
5:  r2 = creat("short1" , 0x1ed)
6:  close(r2)
7:  rename("short1" , "short2" )
8:  close(r0)
```

Exhibit: Buffer Overflow in inotify (CVE-2017-7533)

Crash-inducing mutation

```
1:  mmap(...)
2:  r0 = inotify_init ()
3:  r1 = inotify_add_watch(r0, ".", 0xfff)
4:  chmod(".",  0x1ed)
5:  r2 = creat("short1" , 0x1ed)
6:  close(r2)
7:  rename("short1" , "long_name" )
8:  close(r0)
```

Conclusion

- State-of-the-art OS fuzzers rely on manual rules
  - Hard to scale
- MoonShine scalably generates seeds from traces of real-world programs
  - Lightweight static analysis to track explicit and implicit dependencies
- Discovered 17 new vulnerabilities in Linux kernel

https://github.com/shankarapailoor/moonshine

Getting integrated into syzkaller

Backup Slides

Limitations/Future Work

- Support more OS/Fuzzers
- No multithreaded dependency tracking
- Inter-procedural dependencies
  - Infer that a file must be created from trace
- Multiple distillation strategies
  - distillation without code coverage?

Static Analysis False Positives/Negatives

- False Positives
  - Imprecise pointer analysis
- False Negatives
  - Incomplete AST traversal - function pointers
  - Shared state is not global variable or struct/union field
  - Aliased struct fields get modified
    - char *p = a->v; p[0] = 1

Coverage Breakdown