Drifuzz

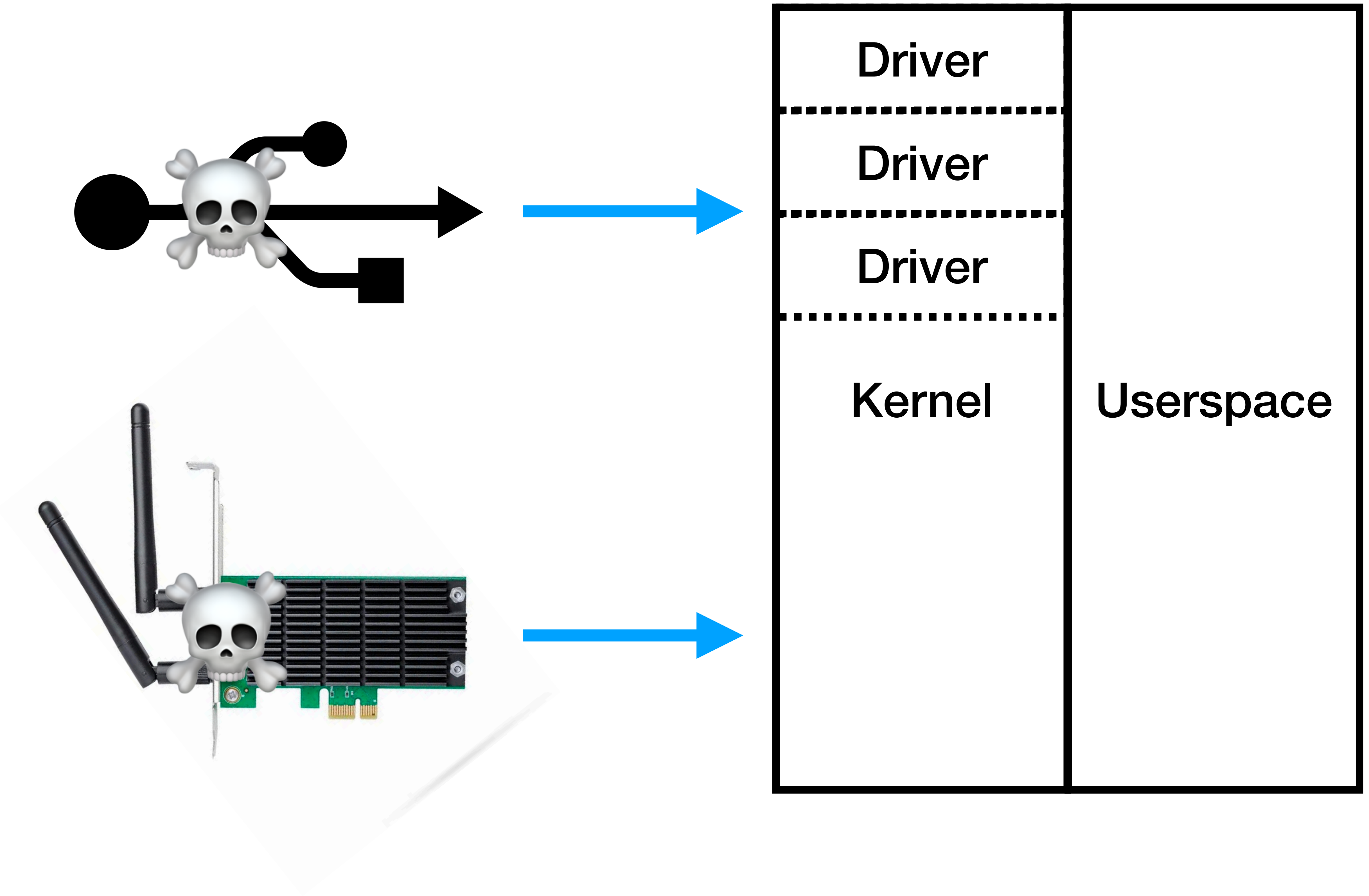
Harvesting Bugs in Device Drivers from Golden Seeds

Published in: USENIX Security 2022
https://messlab.moyix.net/papers/drifuzz_sec22.pdf

Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt

Attack Surface in Device Drivers

- Two major ways for attacker input to reach a driver:
  1. From *userspace,* via ioctl
  2. From the outside world, via a compromised or malicious peripheral
- Traditionally, driver writers have mostly ignored (2)
- Assumed peripherals are “honest” (but maybe flaky/buggy)



Importance of Testing Drivers

- Device drivers are **buggy**: Chou et al. found error rates 3-7x higher than the rest of the kernel [*An empirical study of operating systems errors*, SOSP'01]
- Malicious peripherals can be plugged in via USB, Thunderbolt, etc.
- Modern peripherals are highly complex and run their own (vulnerable) firmware
  - Attacks like **Broadpwn** compromise the WiFi SoC firmware and then exploit bugs in drivers to take over the rest of the system
- Note: older systems gave PCI devices unrestricted access to RAM, making attacks trivial – newer systems use **IOMMU** to restrict access

Challenges of Testing Device Drivers

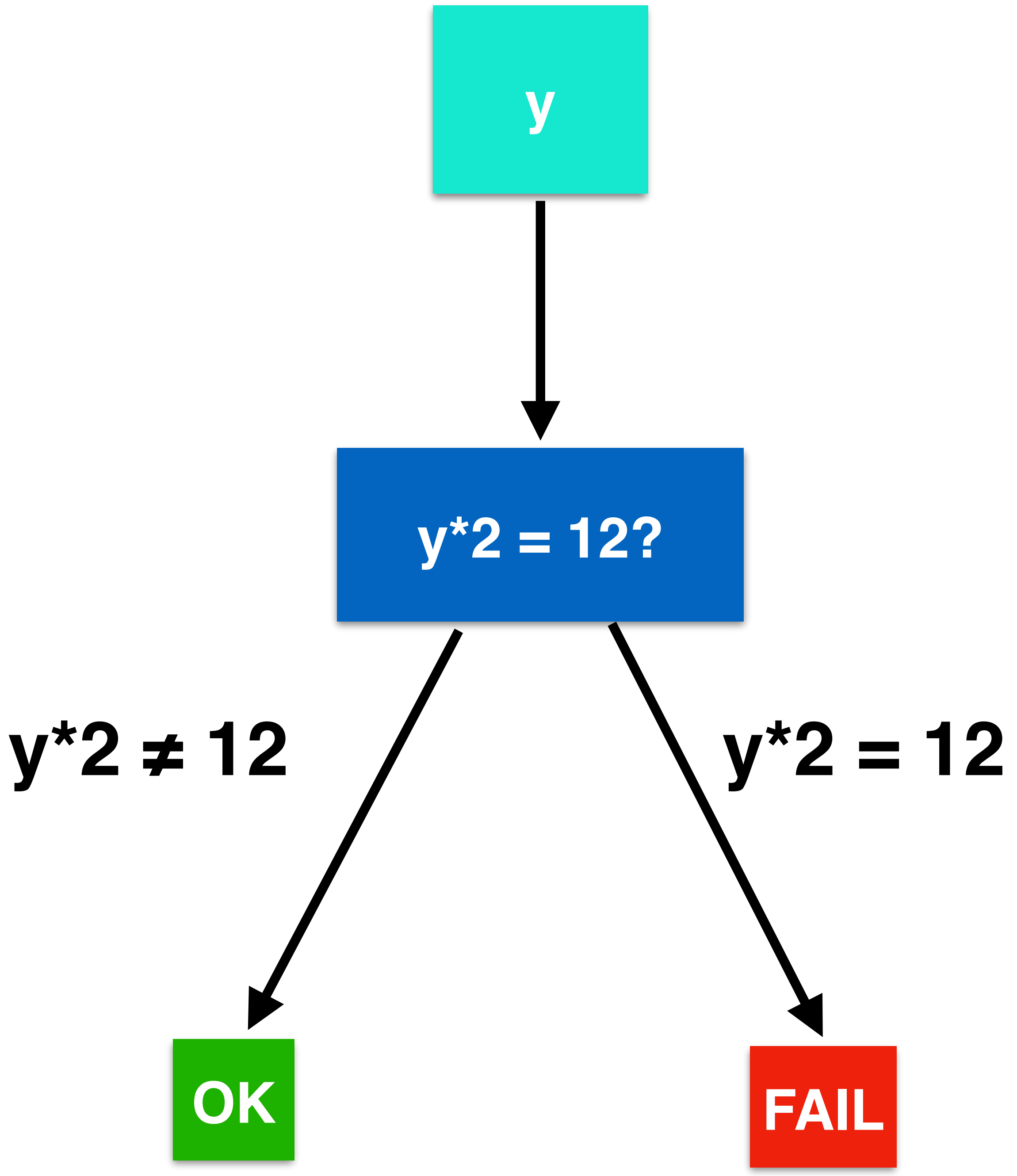
- Lots of different hardware, many different drivers
  - ~14.7 **million** SLoC in Linux kernel's drivers
- Malicious peripherals can pretend to be any of them to target a vulnerable driver
- Impractical to get *real* hardware for all of these!

Emulation: Testing Drivers Without HW

- Can we just emulate peripherals with (e.g.) QEMU?
- Usually **no**: lots of effort needed to create an emulated model for each peripheral
  - Often **more** work than writing a device driver
- Solution: create “dummy” emulated peripherals and then feed inputs to test the device driver
  - Memory-mapped I/O
  - Direct Memory Access (DMA)

Symbolic Execution

- Basic idea: make input *symbolic* and track derived values as *symbolic expressions*
- At a symbolic branch, *fork* the execution and explore both true and false conditions
- The collection of branch conditions for each path can be sent to a constraint solver like Z3 to check satisfiability



OK => (y*2 ≠ 12) => y ≠ 6

FAIL => (y*2 = 12) => y = 6

```
y = read()
y = 2 * y
if (y == 12)
   fail()
print("OK")
```

Concolic Execution

- Concolic execution explores **one** path at a time, starting with a concrete input
- Uses constraint solver to flip individual branches one at a time
- Figure credit: *SAGE: Whitebox Fuzzing for Security Testing*, Godefroid et al. (2012)

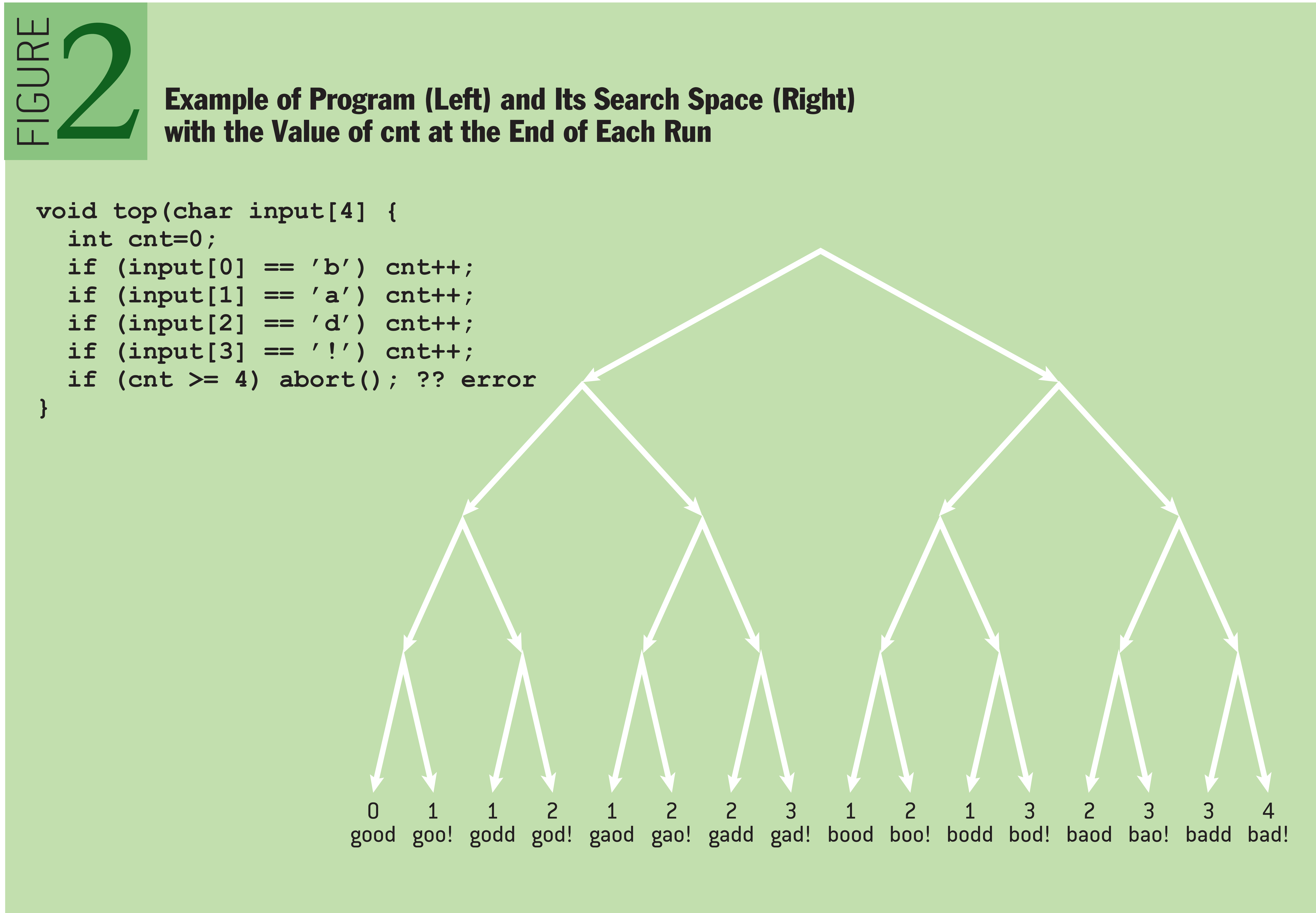


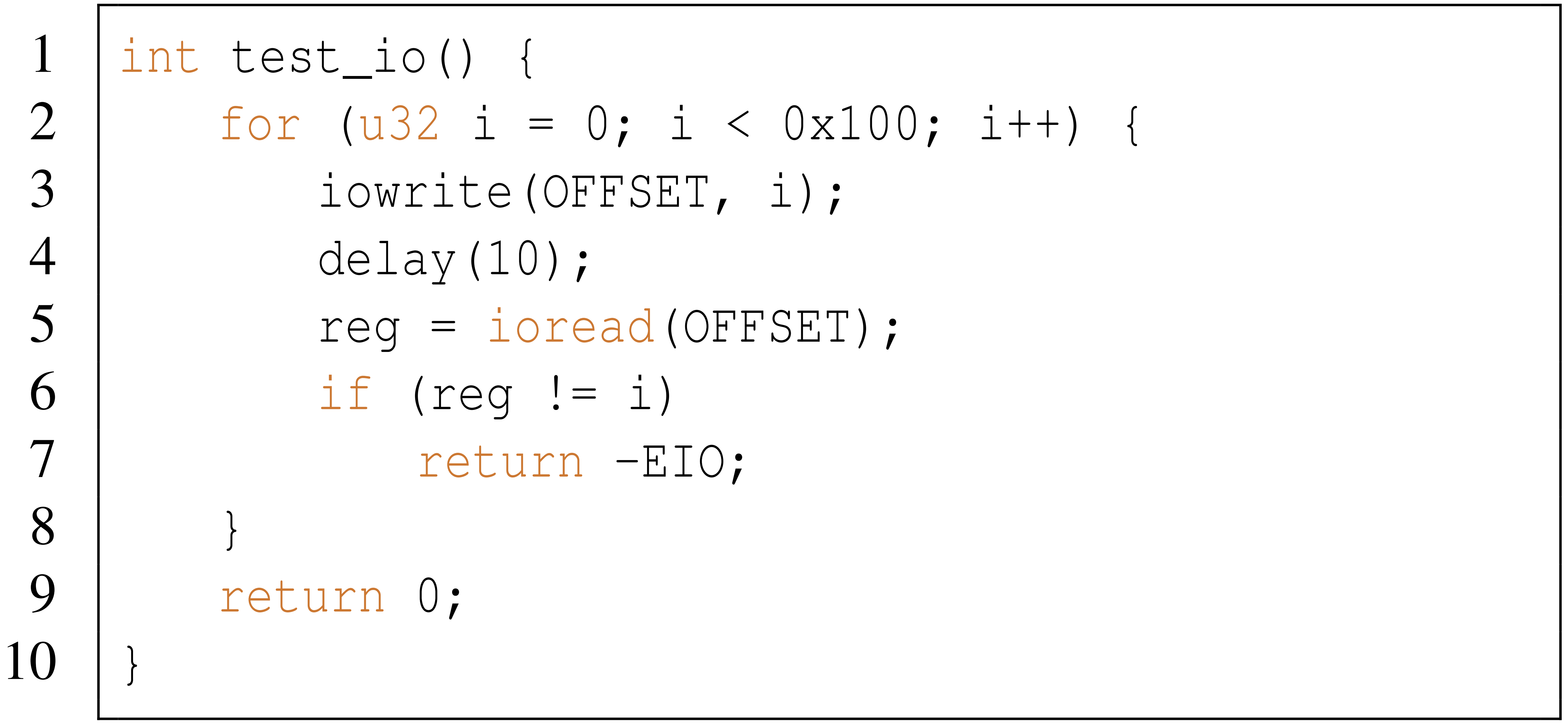
FIGURE 2 **Example of Program (Left) and Its Search Space (Right) with the Value of cnt at the End of Each Run**

```
void top(char input[4] {
  int cnt=0;
  if (input[0] == ’b’) cnt++;
  if (input[1] == ’a’) cnt++;
  if (input[2] == ’d’) cnt++;
  if (input[3] == ’!’) cnt++;
  if (cnt >= 4) abort(); ?? error
}
```

Hard-to-Test Code Patterns

Symbolic Execution

- Symbolic execution has been previously used to test device drivers (SymDrive, 2012)
- But complex drivers (WiFi, Ethernet) contain patterns that make life hard for symbolic execution
- Repetitive loops with symbolic branches can cause **path explosion**



```
int test_io() {
    for (u32 i = 0; i < 0x100; i++) {
        iowrite(OFFSET, i);
        delay(10);
        reg = ioread(OFFSET);
        if (reg != i)
            return -EIO;
    }
    return 0;
}
```

Listing 3: Atheros ath9k driver initialization test code snippet

Fuzzing

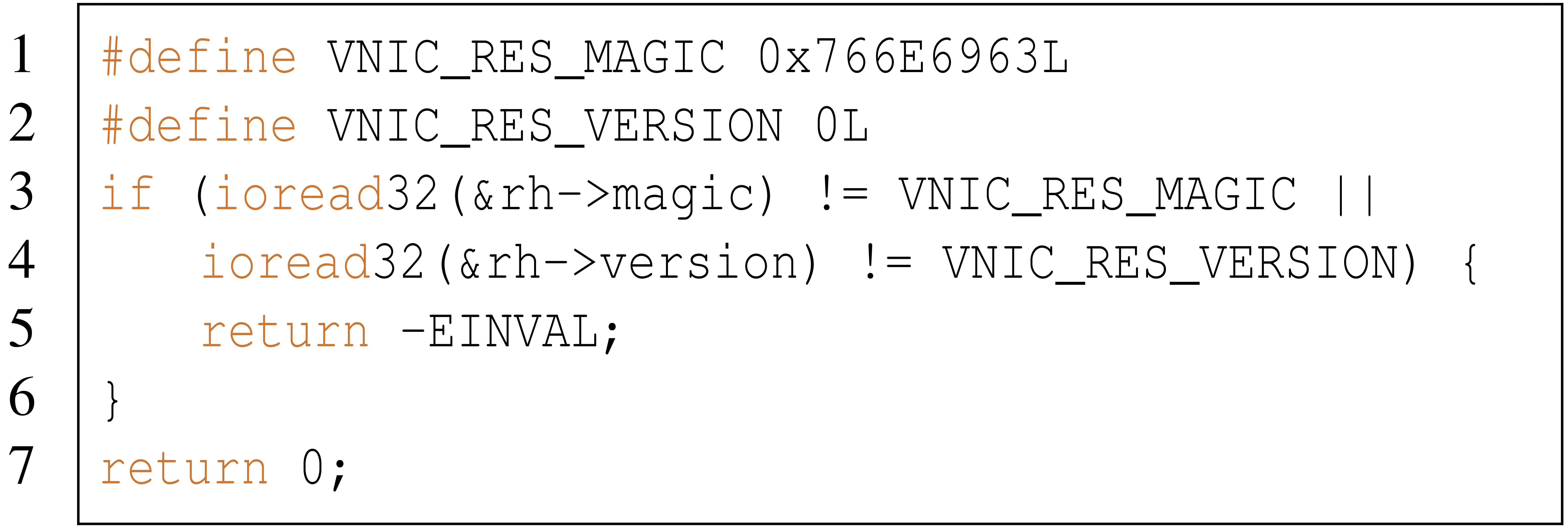
- Another popular technique for software testing in recent years is **fuzzing**
- Popularized by mutational fuzzers like **American Fuzzy Lop (AFL)**
- Starting with some seed inputs, loop:
  - Apply random mutation to inputs
  - Execute the program on each input
  - Measure **coverage** (usually edge coverage)
  - Select inputs that find new coverage

Fuzzing

- Another popular technique for software testing in recent years is **fuzzing**
- Popularized by mutational fuzzers like **American Fuzzy Lop (AFL)**
- Starting with some seed inputs, loop:
  - Apply random mutation to inputs
  - Execute the program on each input
  - Measure **coverage** (usually edge coverage)
  - Select inputs that find new coverage

Hard-to-Test Code Patterns

Fuzzing



```
#define VNIC_RES_MAGIC 0x766E6963L
#define VNIC_RES_VERSION 0L
if (ioread32(&rh->magic) != VNIC_RES_MAGIC ||
    ioread32(&rh->version) != VNIC_RES_VERSION) {
    return -EINVAL;
}
return 0;
```

Listing 1: Magic value check in `snic`.

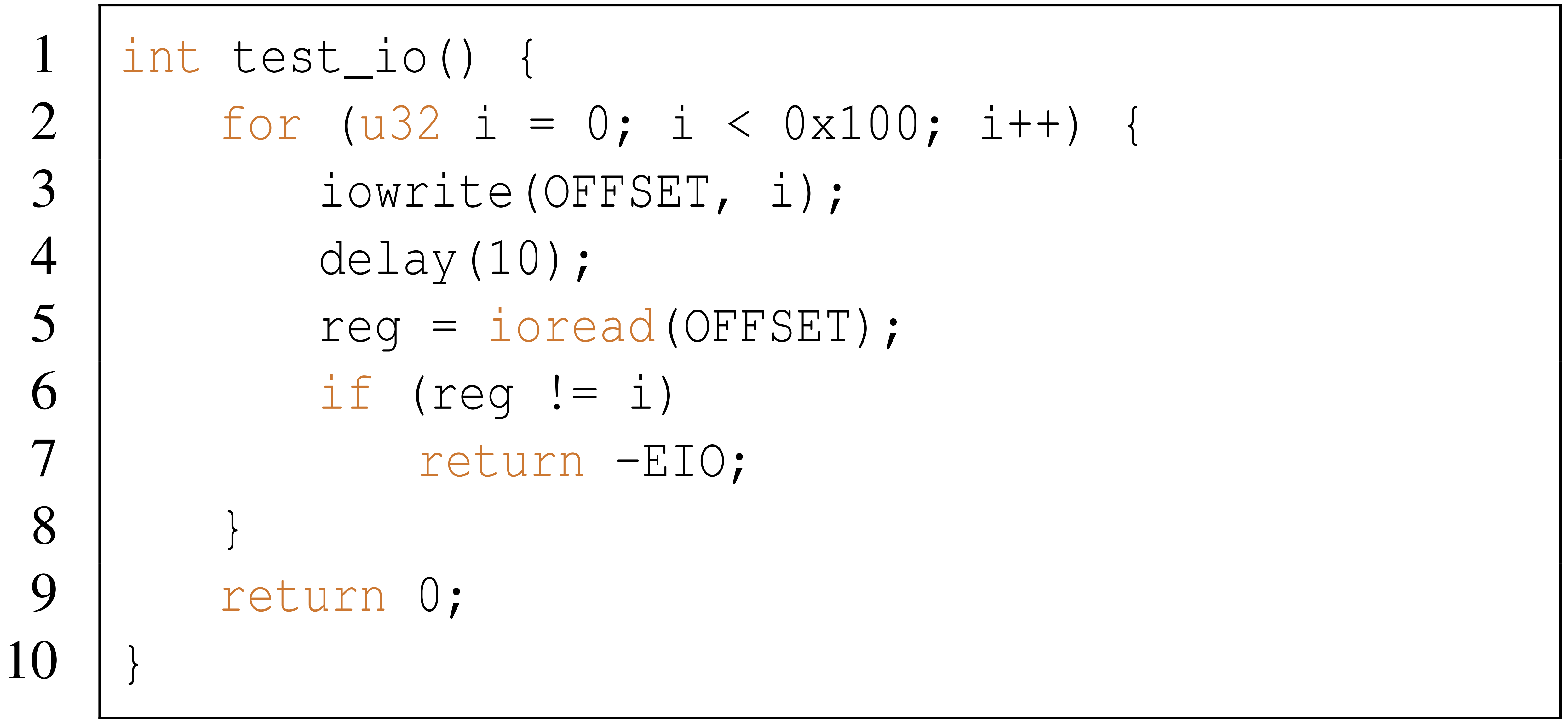
Problem: random mutations have a very hard time guessing magic values!

Golden Seed Generation

- **Key Idea:** Many of these hard-to-test patterns occur during *driver initialization*
    - There is often one “main path” that leads to successful initialization
    - Fuzzers get stuck on hard-to-pass **blocking branches** in this phase
    - If we can find a good **seed** that initializes the driver using more heavyweight techniques like symbolic execution, then we can use it to bootstrap our fuzzing
- Approach: use **concolic execution** to greedily increase the number of symbolic branches covered and learn “**preferred conditions**” for blocking branches
- To help with repetitive loops, use **forced execution** to gather many constraints at once

Optimization: Forced Execution

- Recall our problematic example from before: repetitive check in a loop
- Normal concolic execution would need 256 (0x100) iterations to get past the loop
- We can instead force the branch on **line 6** to always return **false**
- Then collect all the path constraints & solve with a single iteration

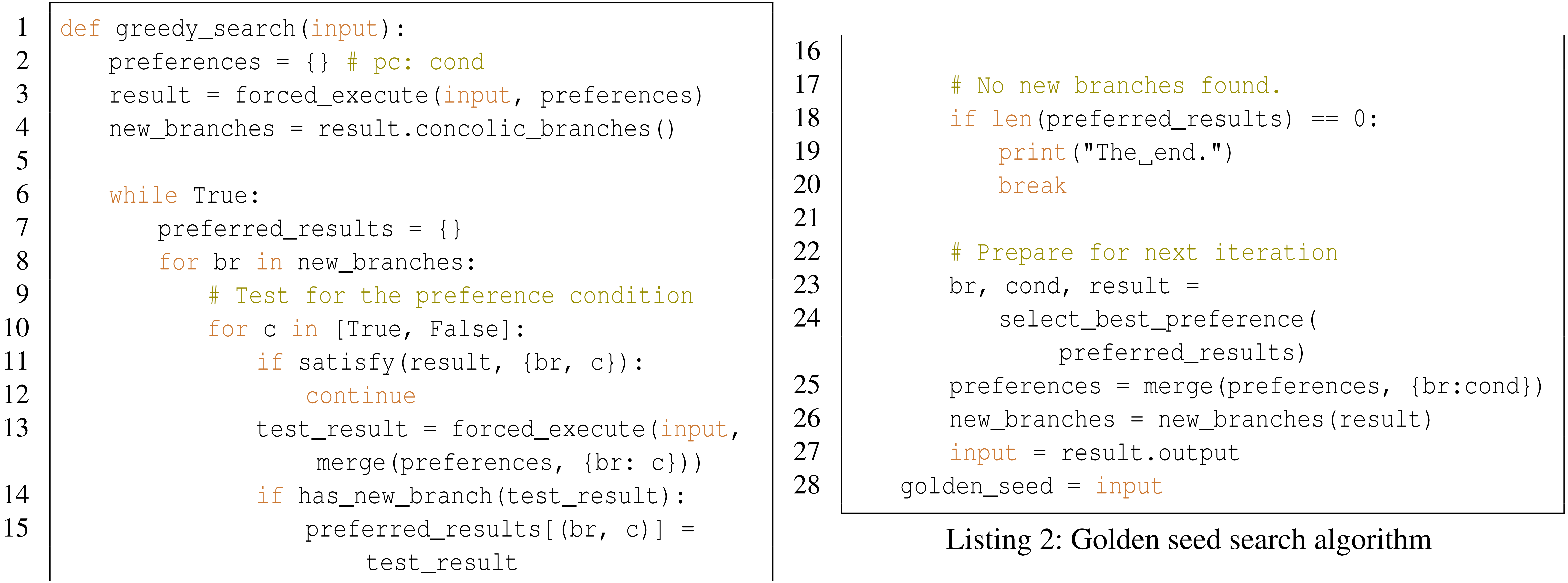


```
1  int test_io() {
2      for (u32 i = 0; i < 0x100; i++) {
3          iowrite(OFFSET, i);
4          delay(10);
5          reg = ioread(OFFSET);
6          if (reg != i)
7              return -EIO;
8      }
9      return 0;
10 }
```

Listing 3: Atheros ath9k driver initialization test code snippet

- **NB**: This can lead to infeasible path constraints! But works well in practice.

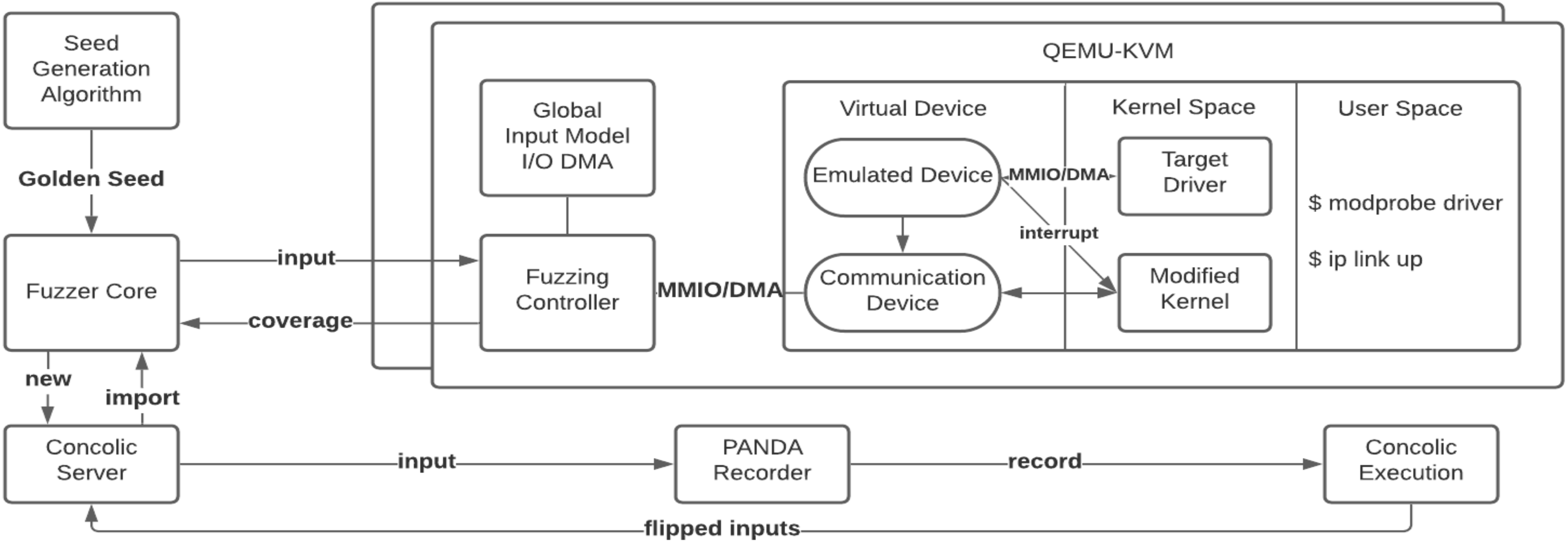
Golden Seed Generation Algorithm



```
def greedy_search(input):
    preferences = {} # pc: cond
    result = forced_execute(input, preferences)
    new_branches = result.concolic_branches()

    while True:
        preferred_results = {}
        for br in new_branches:
            # Test for the preference condition
            for c in [True, False]:
                if satisfy(result, {br, c}):
                    continue
                test_result = forced_execute(input,
                     merge(preferences, {br: c}))
                if has_new_branch(test_result):
                    preferred_results[(br, c)] =
                         test_result

        # No new branches found.
        if len(preferred_results) == 0:
            print("The␣end.")
            break

        # Prepare for next iteration
        br, cond, result =
            select_best_preference(
                 preferred_results)
        preferences = merge(preferences, {br:cond})
        new_branches = new_branches(result)
        input = result.output
    golden_seed = input
```

Listing 2: Golden seed search algorithm

Drifuzz System Design

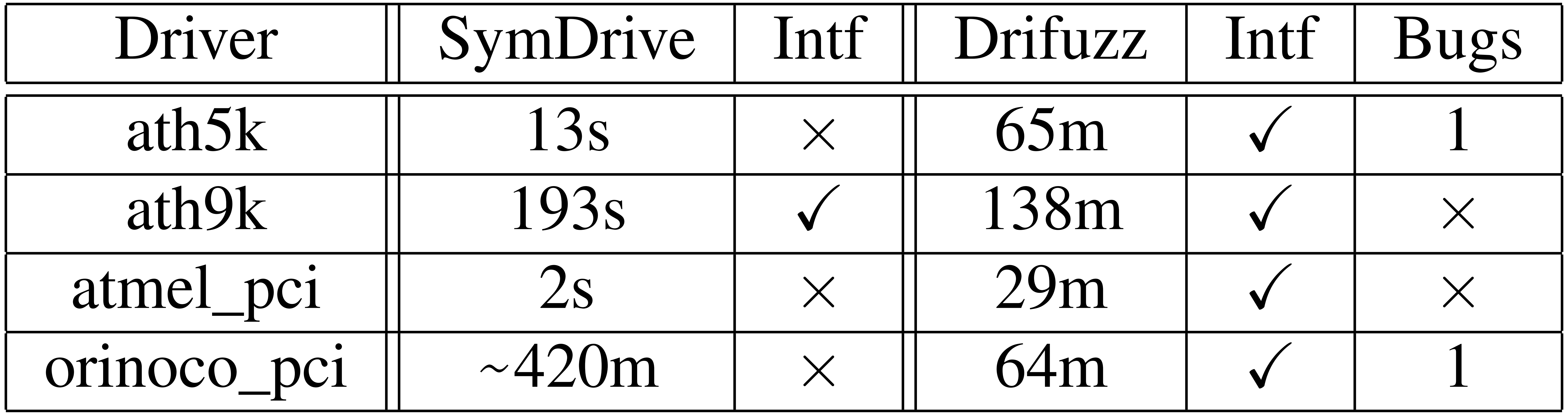


Implementation

- Golden seed search implemented using **PANDA** dynamic analysis platform (https://panda.re)
- PANDA supports dynamic taint analysis by lifting binary code to LLVM (via S2E), supports whole-system record/replay
- We added concolic execution support by having taint system track Z3 symbolic exprs
- Fuzzing component extends previous KVM-based fuzzer, kAFL

Component	Lines
Linux Comm Driver and DMA Tracking	470 + 0
PANDA Concolic Support	842 + 77
PANDA Customization	2421 + 146
Fuzzing Backend (adapted from kAFL)	872 + 331
Fuzzing Scripts	874 + 0
Concolic Scripts	2721 + 0

Evaluation: Comparison with SymDrive

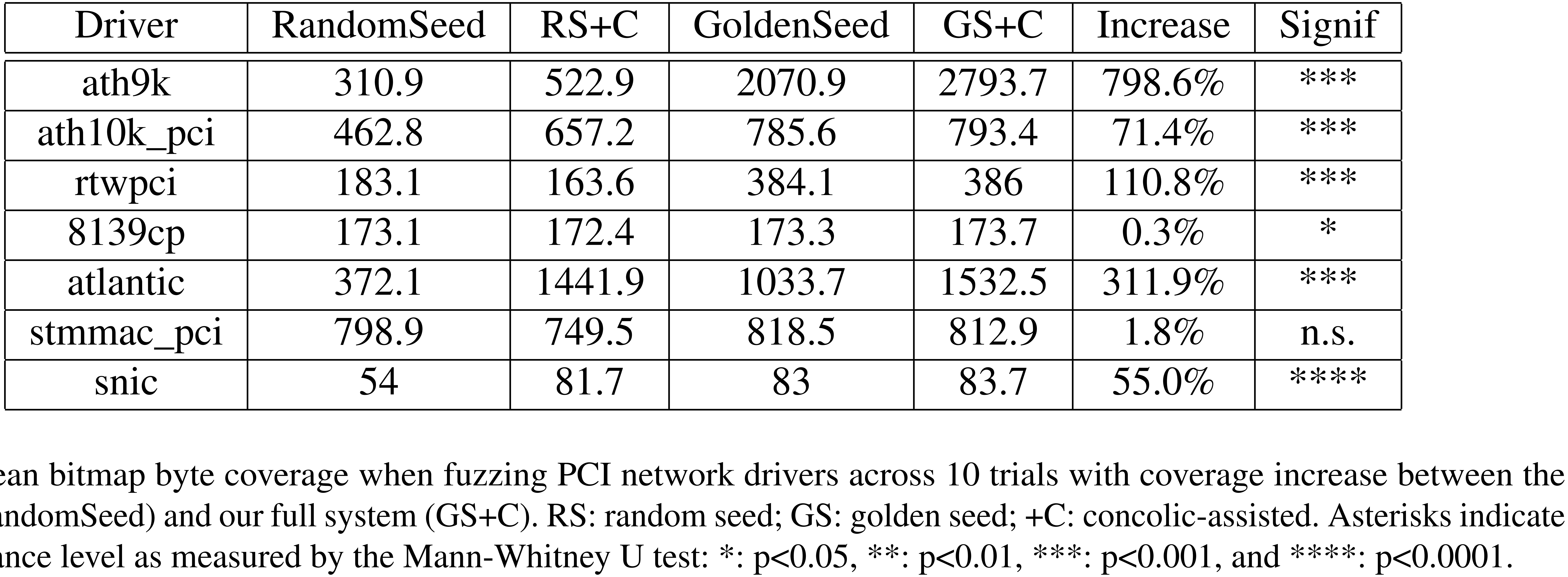


Driver	SymDrive	Intf	Drifuzz	Intf	Bugs
ath5k	13s	×	65m	✓	1
ath9k	193s	✓	138m	✓	×
atmel_pci	2s	×	29m	✓	×
orinoco_pci	~420m	×	64m	✓	1

- Evaluation somewhat limited — SymDrive is 10 years old, had to backport Drifuzz to Linux 3.1.1 and add configs for some WiFi drivers
- Evaluation tests bugs found & whether network interface is initialized
- **Result:** SymDrive usually completes more quickly, but can get stuck due to path explosion often does not successfully initialize interface
- Drifuzz also finds two bugs, one of which was still unfixed in current Linux

Evaluation: Ablation

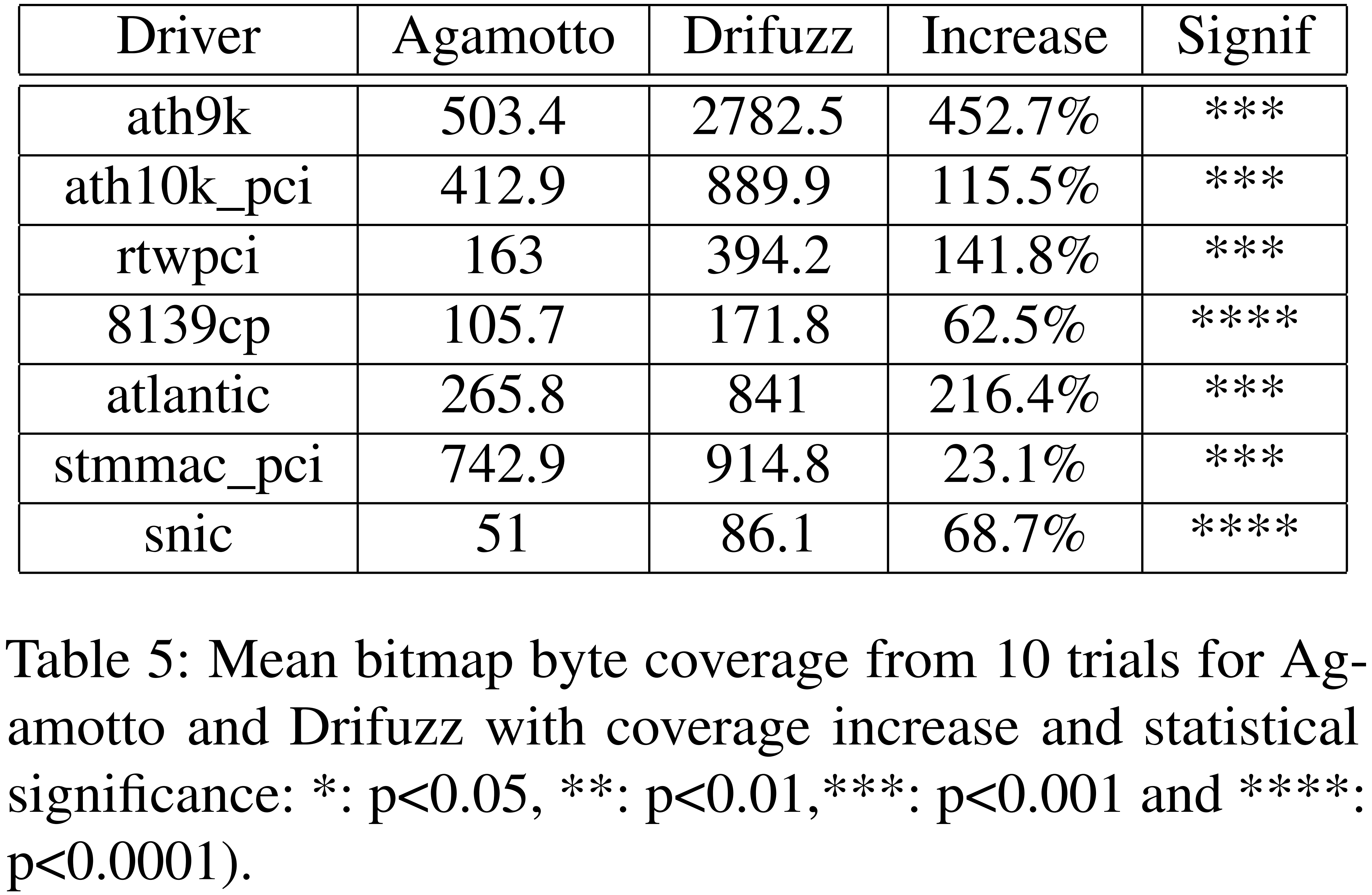
How do different components contribute?



Driver	RandomSeed	RS+C	GoldenSeed	GS+C	Increase	Signif
ath9k	310.9	522.9	2070.9	2793.7	798.6%	***
ath10k_pci	462.8	657.2	785.6	793.4	71.4%	***
rtwpci	183.1	163.6	384.1	386	110.8%	***
8139cp	173.1	172.4	173.3	173.7	0.3%	*
atlantic	372.1	1441.9	1033.7	1532.5	311.9%	***
stmmac_pci	798.9	749.5	818.5	812.9	1.8%	n.s.
snic	54	81.7	83	83.7	55.0%	****

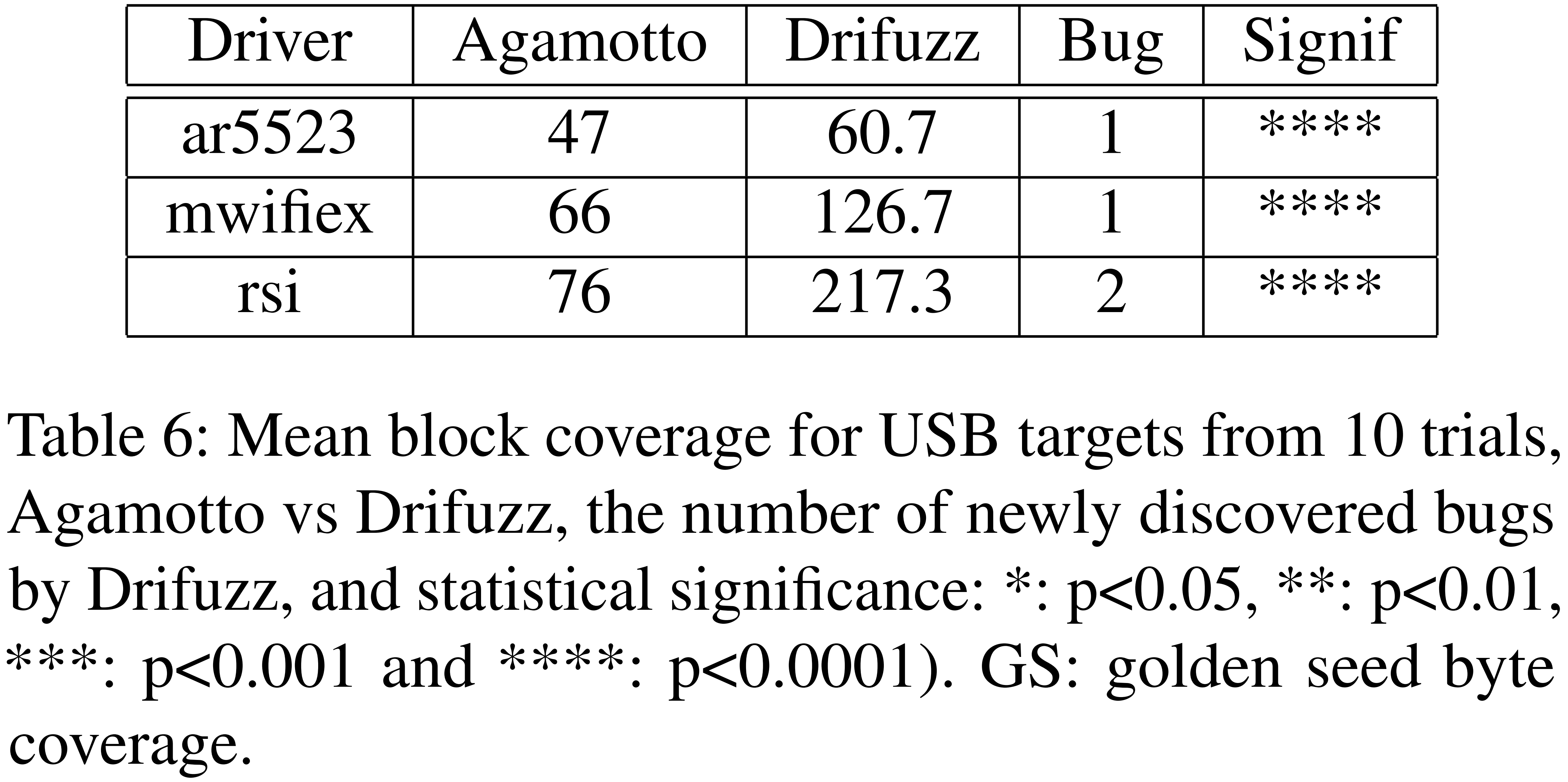
Table 3: Mean bitmap byte coverage when fuzzing PCI network drivers across 10 trials with coverage increase between the baseline (RandomSeed) and our full system (GS+C). RS: random seed; GS: golden seed; +C: concolic-assisted. Asterisks indicate the significance level as measured by the Mann-Whitney U test: *: $p<0.05$, **: $p<0.01$, ***: $p<0.001$, and ****: $p<0.0001$.

Evaluation: Comparison with Agamotto



Driver	Agamotto	Drifuzz	Increase	Signif
ath9k	503.4	2782.5	452.7%	***
ath10k_pci	412.9	889.9	115.5%	***
rtwpci	163	394.2	141.8%	***
8139cp	105.7	171.8	62.5%	****
atlantic	265.8	841	216.4%	***
stmmac_pci	742.9	914.8	23.1%	***
snic	51	86.1	68.7%	****

Table 5: Mean bitmap byte coverage from 10 trials for Agamotto and Drifuzz with coverage increase and statistical significance: *: p<0.05, **: p<0.01,***: p<0.001 and ****: p<0.0001).



Driver	Agamotto	Drifuzz	Bug	Signif
ar5523	47	60.7	1	****
mwifiex	66	126.7	1	****
rsi	76	217.3	2	****

Table 6: Mean block coverage for USB targets from 10 trials, Agamotto vs Drifuzz, the number of newly discovered bugs by Drifuzz, and statistical significance: *: p<0.05, **: p<0.01, ***: p<0.001 and ****: p<0.0001). GS: golden seed byte coverage.

Evaluation: Bug-Finding

Summary	Driver	Type	Fixed	Stage
KASAN: slab-out-of-bounds in ath10k_pci_hif_exchange_bmi_msg	ath10k	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in hw_atl_utils_fw_upload_dwords	atlantic	PCI	✓	fuzzing
KASAN: double-free or invalid-free in consume_skb	atlantic	PCI	✓	seed-gen
KASAN: use-after-free in stmmac_napi_poll_rx	stmmac	PCI	✓	seed-gen
KASAN: use-after-free in aq_ring_rx_clean	atlantic	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in ath5k_eeprom_read_pcal_info_5111	ath5k	PCI	✓	seed-gen
KASAN: null-ptr-deref	ar5523	USB	✓	seed-gen
skbuff: skb_over_panic	mwifiex	USB	✓	seed-gen
KASAN: slab-out-of-bounds in ath9k_hif_usb_rx_cb	ath9k_htc	USB	✓	seed-gen
KASAN: slab-out-of-bounds in rsi_read_pkt	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_rx_done_handler	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_read_pkt	rsi	USB		fuzzing

Vulnerabilities Found

- Two of the bugs found by Drifuzz were considered serious enough to warrant CVE identifiers
- **CVE-2021-43975** is an out-of-bounds read followed by an out-of-bound write with attacker-controlled length in the `atlantic` PCI Ethernet driver
- **CVE-2021-43976** is a kernel panic (denial of service) in the Marvell mwifiex USB driver
- Vulnerabilities + patches were reported via LKML, we worked with downstream distro to help understand impact

Conclusions

- Testing device drivers is still difficult!
  - Limited hardware availability
  - Complex driver conditions & tests
  - Slow execution speeds (whole-system VM)
- Drifuzz's golden seeds can make testing much more efficient and effective
  - Golden seeds can also be re-used as good starting points for other driver testing techniques
- Check it out! https://github.com/messlabnyu/DrifuzzProject

We are currently working on this one :)