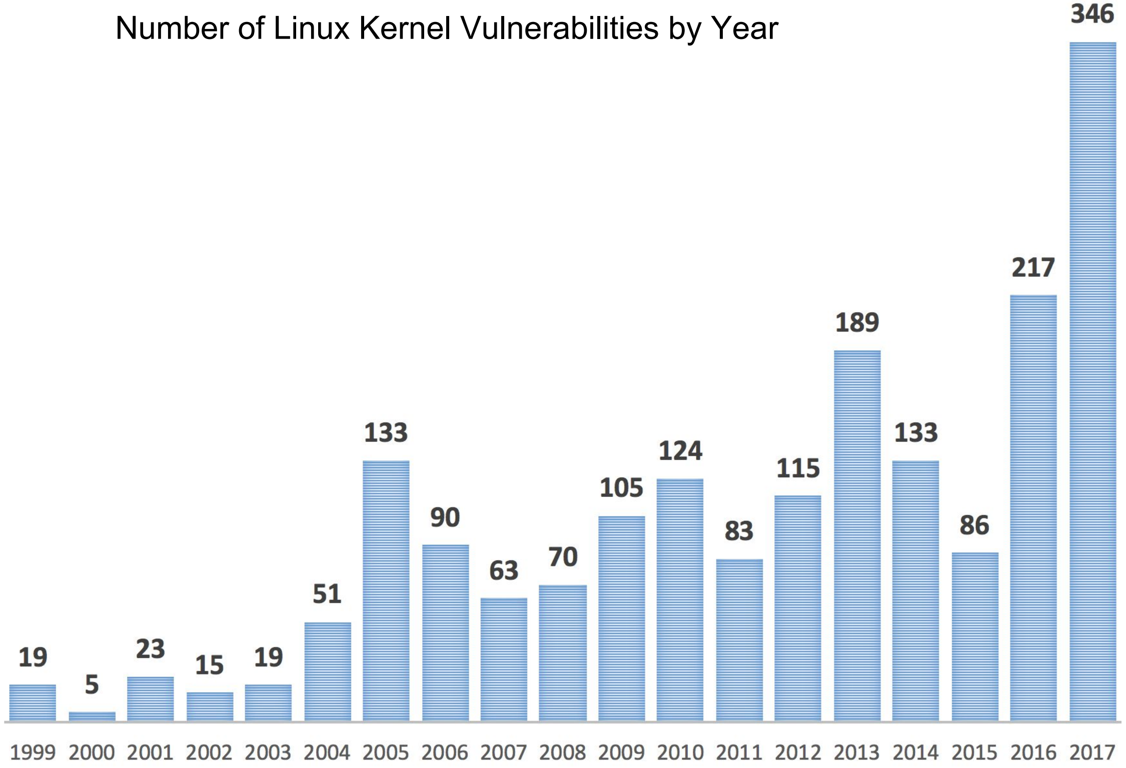
Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path

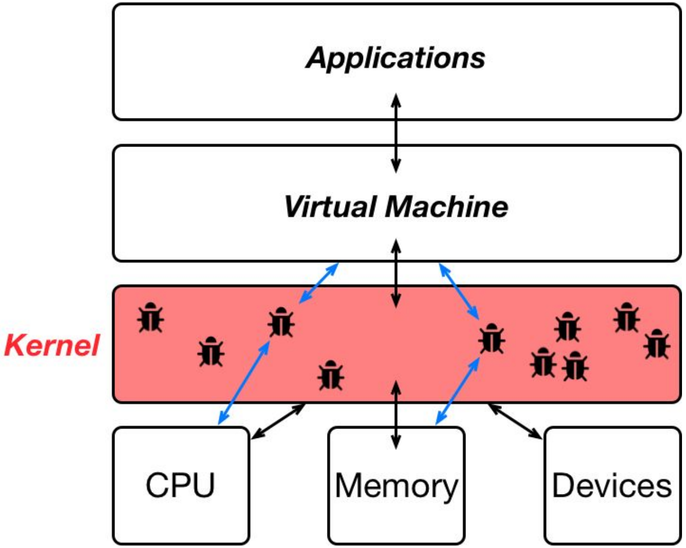
Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, Justin Cappos

New York University
Tandon School of Engineering

Motivation

1. Many vulnerabilities exist in the host OS kernel
2. These vulnerabilities can be reached and exploited, even with VMs in place

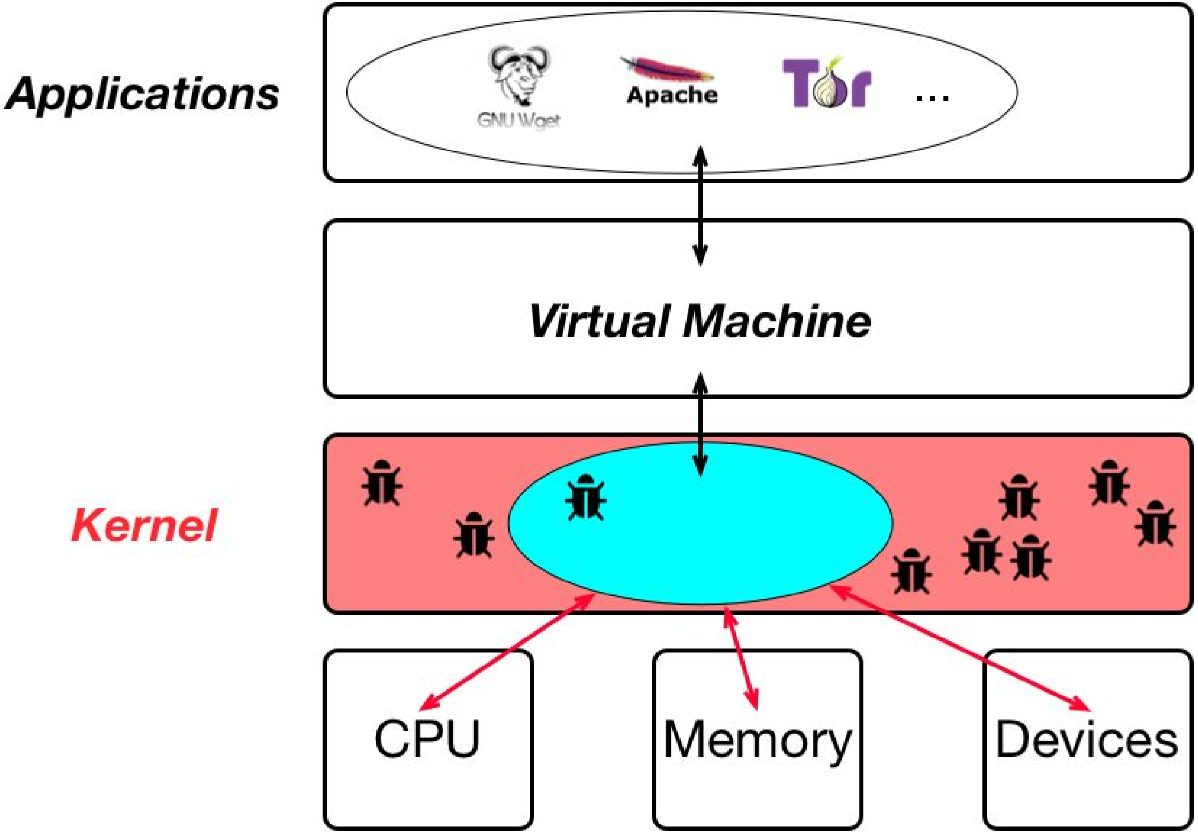




* Data source: National Vulnerability Database(NVD), https://nvd.nist.gov, July 2017.

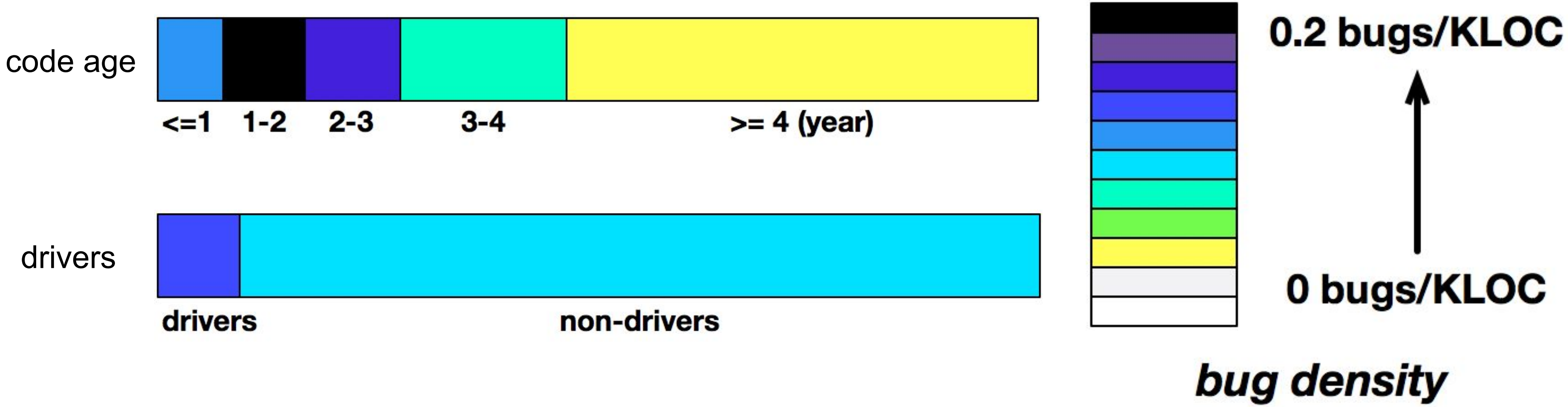
What do we want when building virtual machines?

1. Sufficient functionality
2. Very few zero-day security bugs



The metrics we have don’t meet our needs

1. Predivtive of where bugs will be found
2. Locate areas that have no/very few bugs



code age [1]

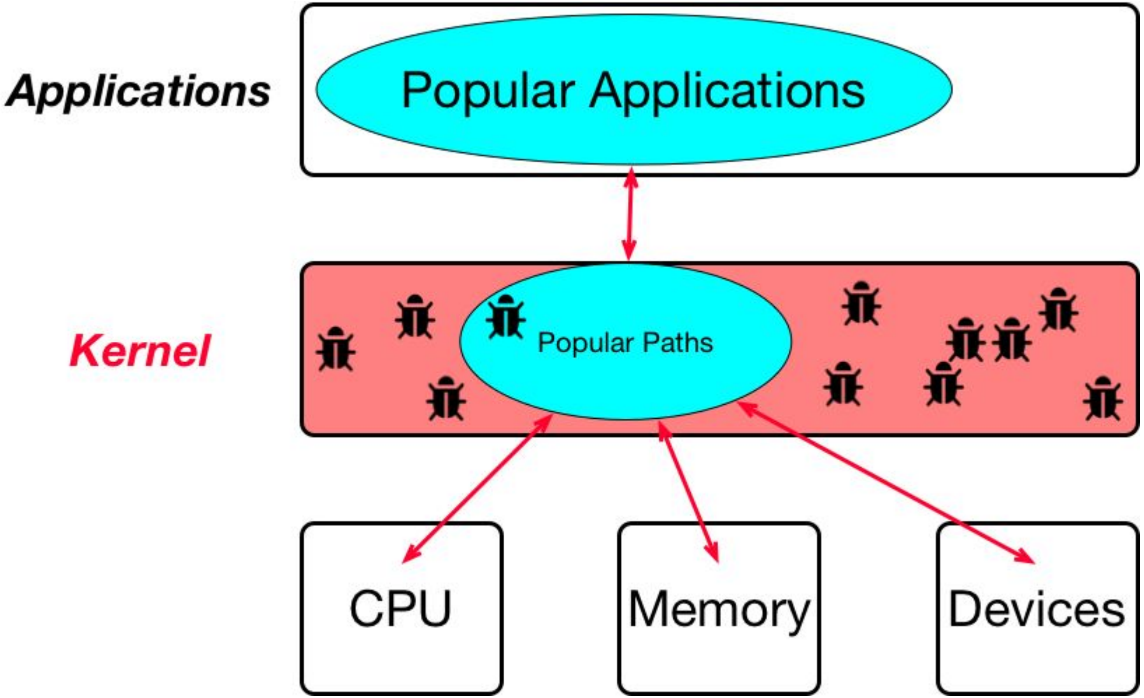
code in device drivers [2]

[1] Ozment, et al. [Usenix Security ’06]

[2] Chou, et al. [SOSP ’01]

Our metric: the popular paths

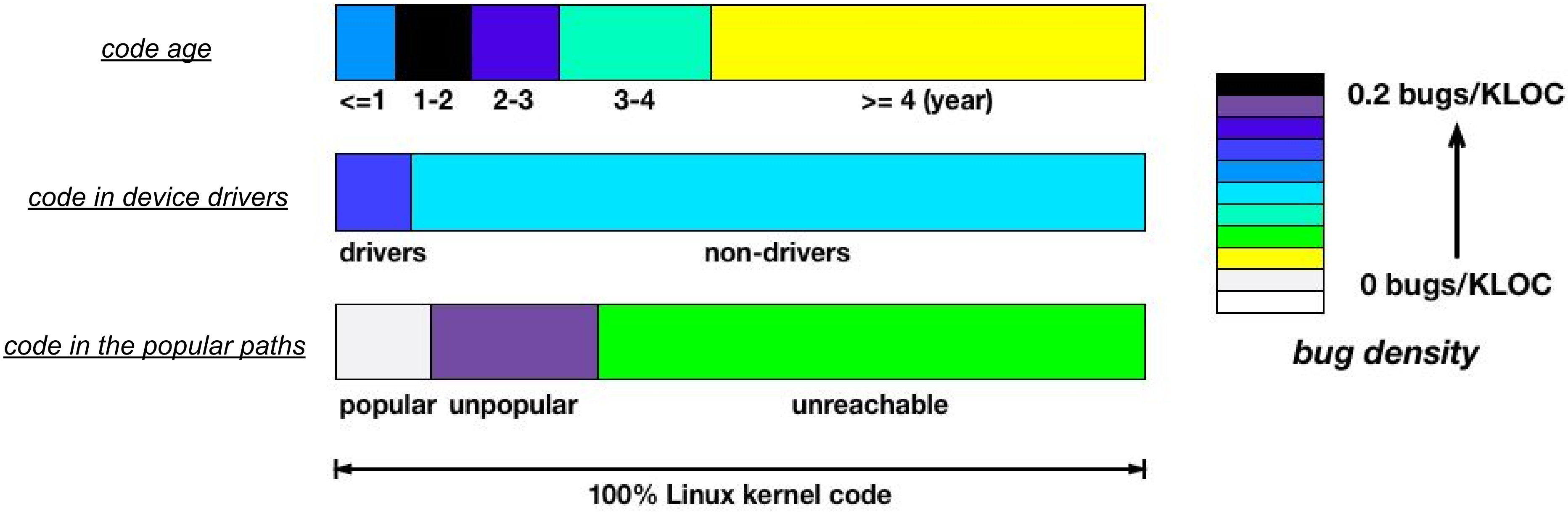
- ***Definition:*** lines of code in the kernel source files, which are commonly executed in the system's normal workload.
- ***Key insight:*** the popular paths contain many fewer bugs!



Our experiments to obtain the popular paths

- Ran top 50 most popular packages according to the Debian popularity contest.
- Two students used their Ubuntu systems for five days.
- We used Gcov 4.8.4 in Ubuntu 14.04 to capture the kernel coverage data.

Bug density comparison among three security metrics



code age [1]

code in device drivers [2]

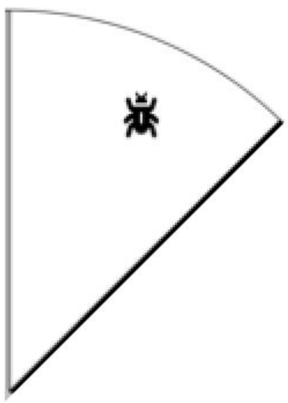
code in the popular paths [3]

[1] Ozment, et al. [Usenix Security ’06]

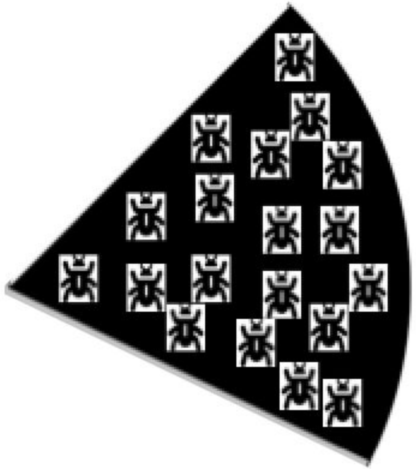
[2] Chou, et al. [SOSP ’01]

[3] Li, et al. [USENIX ATC ’17]

popular paths vs. unpopular paths



popular paths
(1 bug)

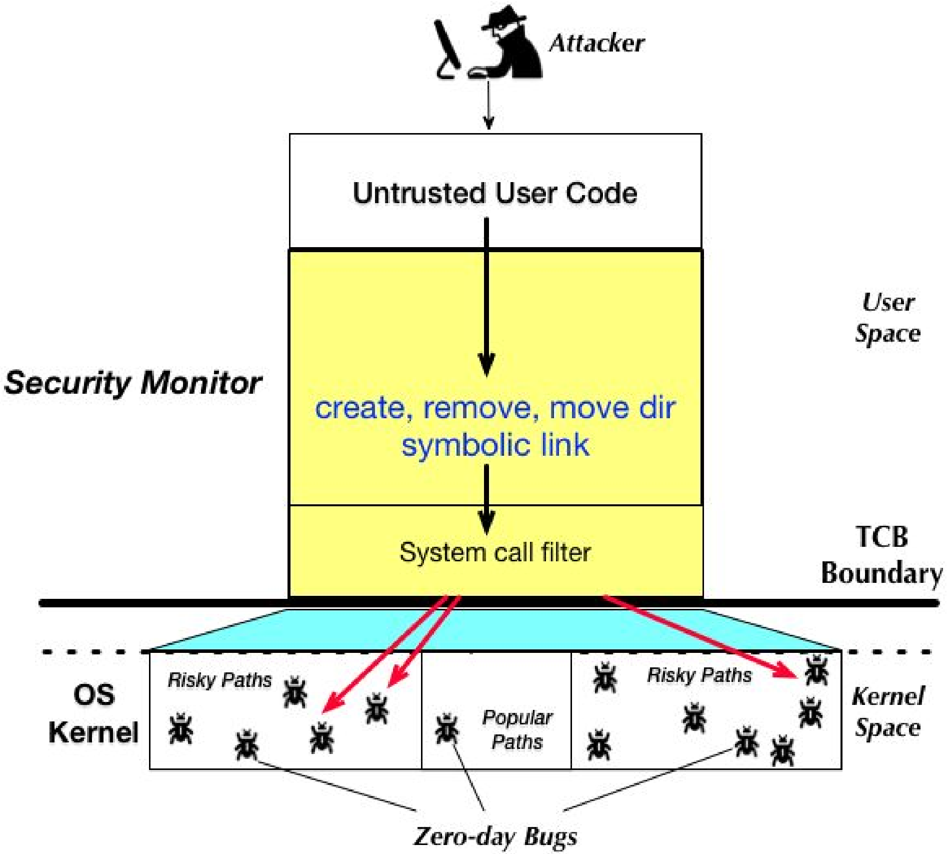


unpopular paths
(19 bugs)

Our metric: the popular paths

- Definition ✓
- How to measure it? ✓
- Is it a good security metric? ✓
- Is it practically useful?

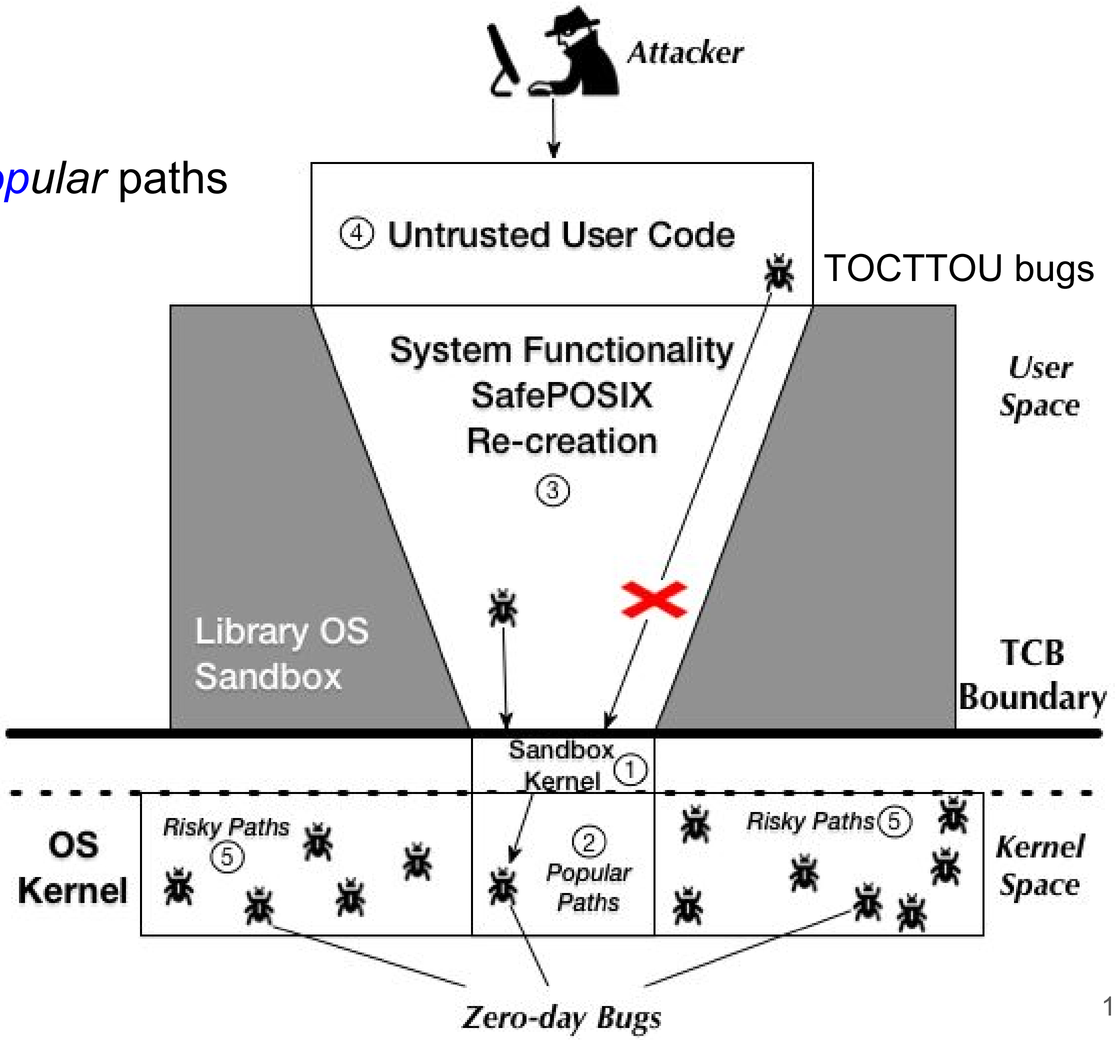
Traditional designs: *check-and-pass-through*



Lock-in-Pop design

lock applications *into* using only *popular* paths

- safely re-create file directories with basic calls like open(), read(), write(), close() to avoid using unpopular paths
- the kernel is used infrequently
- only the popular paths in the kernel is accessed

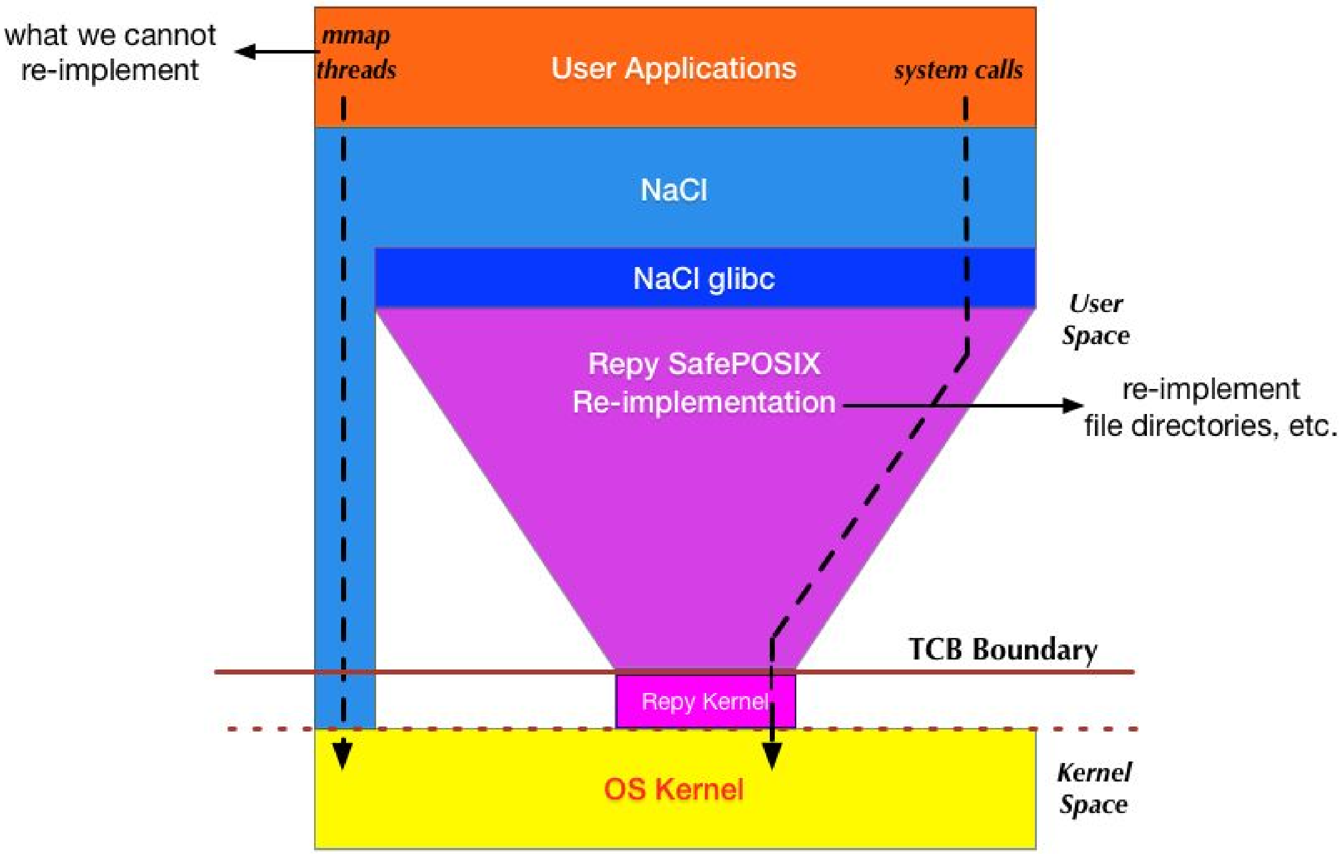


Our prototype implementation: Lind

- Google’s Native Client (NaCl) [IEEE S&P ’09]: software fault isolation

- Repy Sandbox [CCS ’10]
  - Small sandbox kernel (8K LOC)
  - 33 basic API functions
  - Accessed only a subset of the “popular paths”
  - Real-world deployment in the Seattle project, under security audit for 5+ years

Our prototype implementation: Lind



Evaluation results: Linux kernel coverage by fuzzing

Virtualization system	# of bugs	Kernel trace (LOC)		
		Total coverage	In popular paths	In risky paths
LXC	12	127.3K	70.9K	56.4K
Docker	8	119.0K	69.5K	49.5K
Graphene	8	95.5K	62.2K	33.3K
Lind	1	70.3K	70.3K	0
Repy	1	74.4K	74.4K	0

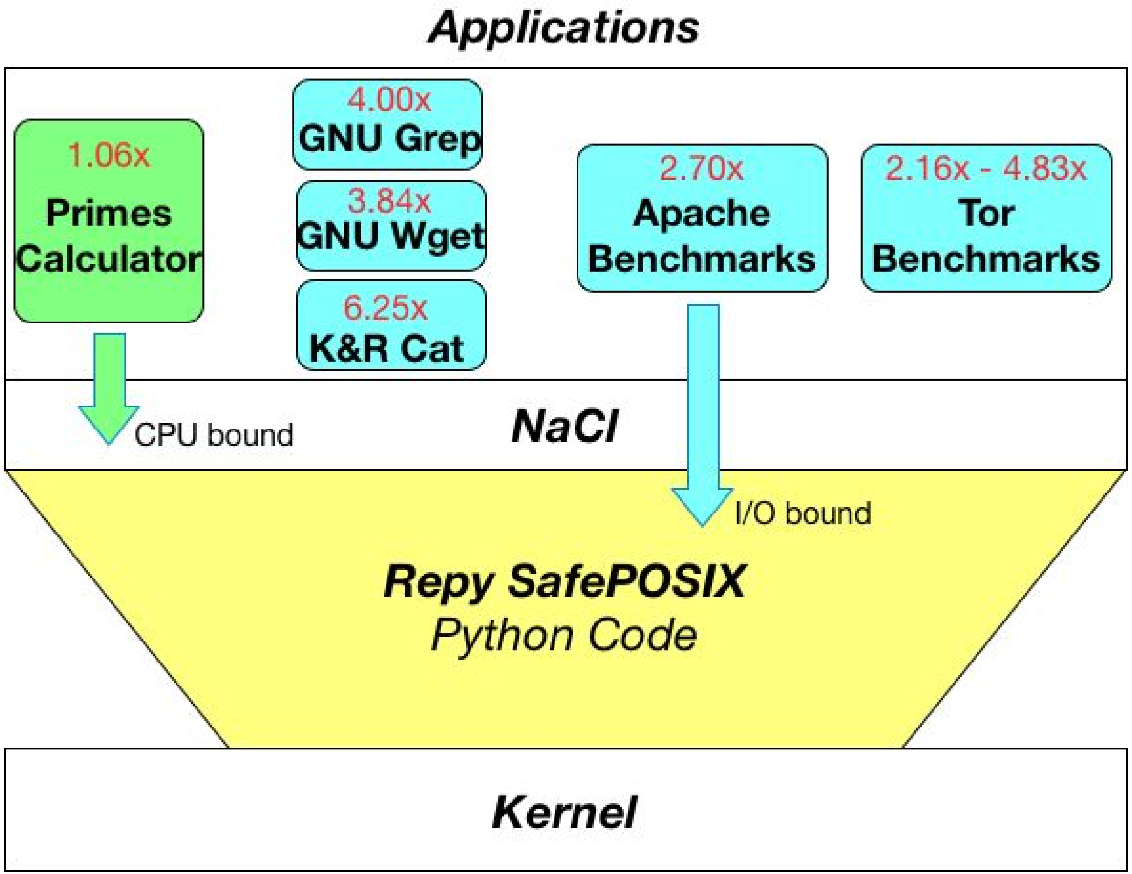
Evaluation results: Linux kernel bugs triggered

VM	**Bugs Triggered**
Native Linux	35/35 (100%)
LXC	12/35 (34.3%)
Docker	8/35 (22.9%)
Graphene	8/35 (22.9%)
Lind	1/35 (2.9%)

Example: CVE-2015-5706, a bug triggered everywhere except Lind

- A rarely-used flag O_TMPFILE reached unpopular lines of code inside fs/namei.c
- Lind is not affacted, because it is avoiding unpopular paths by restricting flags

Evaluation results: performance overhead in Lind



Limitations

- Some bugs are difficult to evaluate using our metric.
- Reaching lines of code may not be sufficient to trigger or exploit a bug.
- Lind’s performance could be improved.

Future work

- Removing risky lines from the kernel.
- Build a minimal OS kernel for Docker’s LinuxKit, etc.

Conclusion

- The popular paths, contain many fewer bugs.
- *Lock-in-Pop* design
- Our prototype system, Lind, exposes fewer zero-day kernel bugs.


Thank You


Q & A