

Improving the Reliability of Commodity Operating Systems

Mike Swift

University of Wisconsin

Hank Levy⁺, Brian Bershad⁺, Muthu
Annamalai^{*}

⁺University of Washington, ^{*}Microsoft

High Level OS Problems

1. Performance
2. Features
3. Security
4. Reliability

Previous Approaches to Reliability

1. Fix the code

[MetaCompilation, Windows Driver Verifier]

2. Build a new system

[Singularity, Tandem NonStop, QuickSilver]

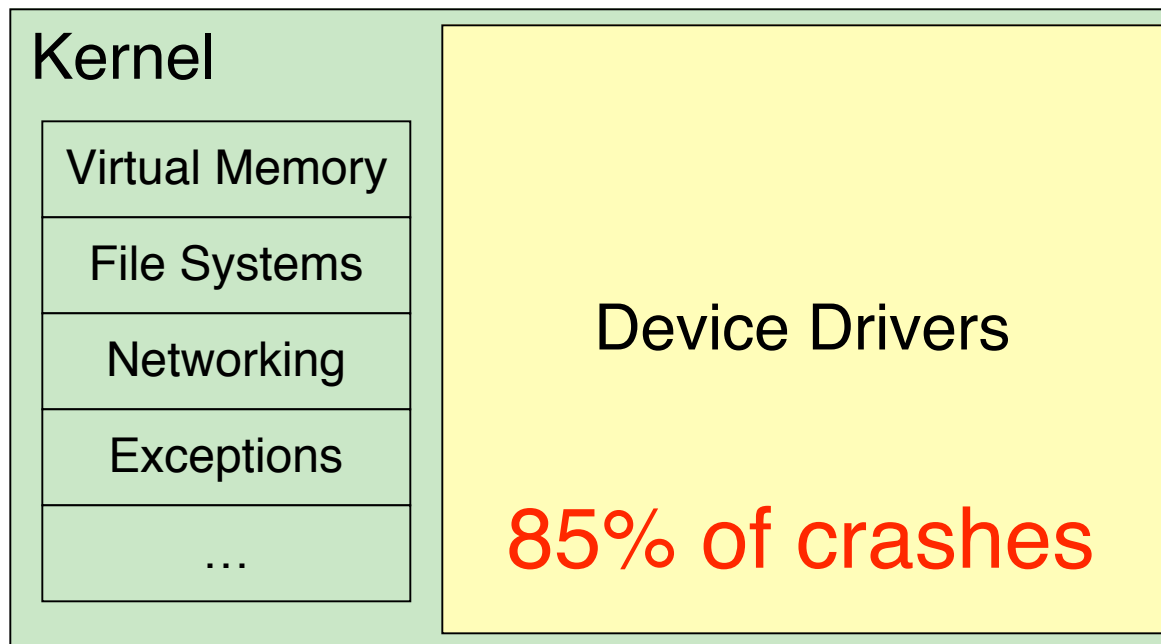
3. Add hardware

[VaxClusters, Wolfpack, Google]

OS Today

Application

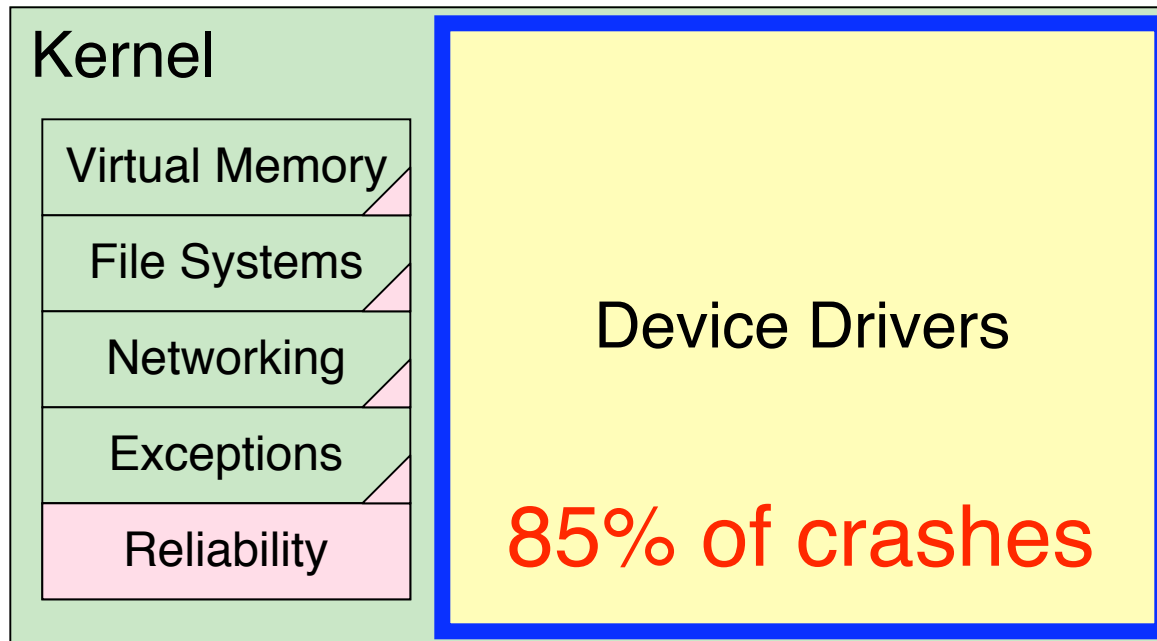
Application



OS With Reliability

Application

Application



Contributions

We designed and built a new kernel subsystem that:

- Prevents majority of driver-caused crashes
- Requires **no** changes to existing drivers
- Requires only minor changes to OS
- Minimally impacts performance

Outline

- Introduction
- **Problem**
- Design
- Evaluation
- Summary

What Is a Driver?

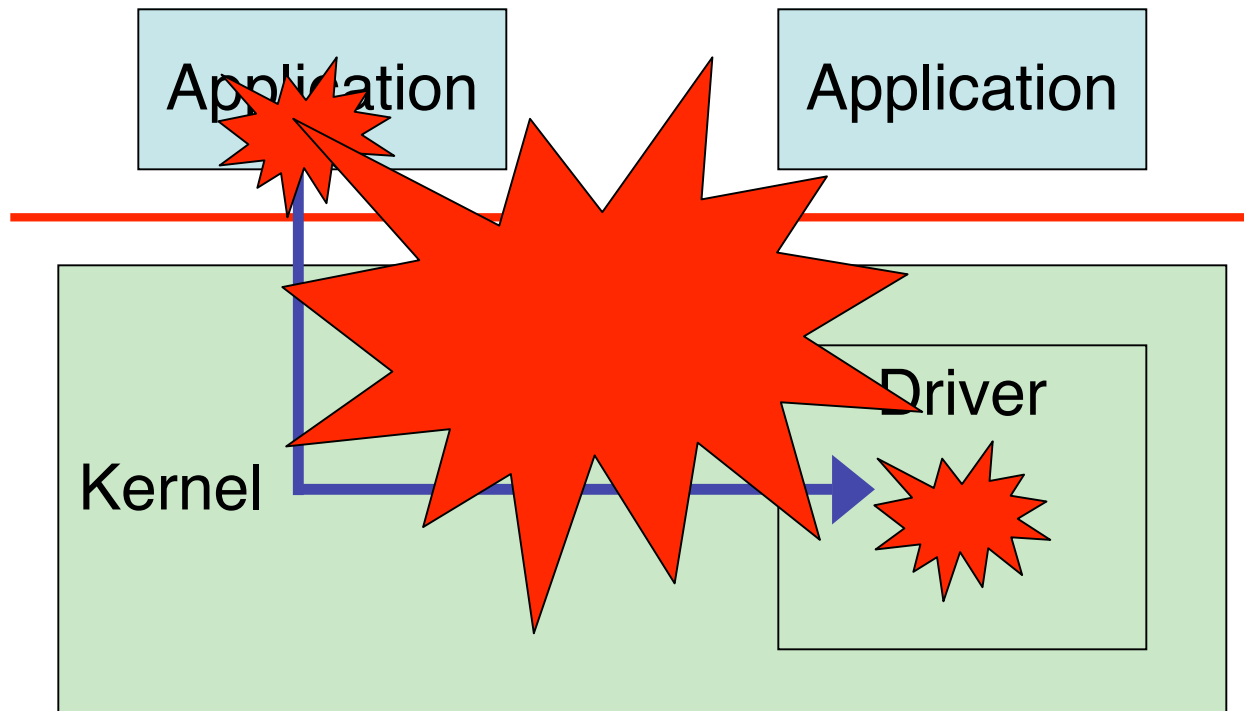
A module that translates OS requests to device requests

- 10s of thousands exist
- 81 drivers running on this laptop!
- Run in the OS kernel
- Small # of common interfaces

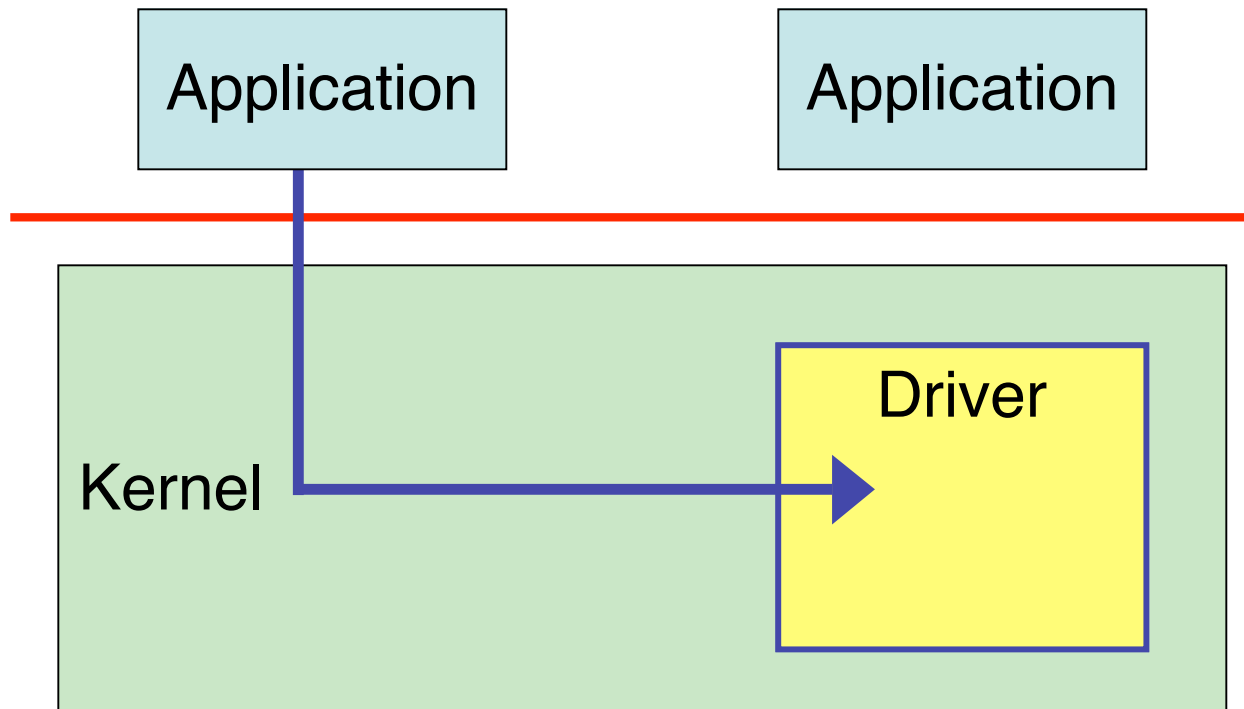
Why Do Drivers Fail?

- Complex and hard to write
 - Must handle asynchronous events
 - Must obey kernel programming rules
 - Non-reproducible failures
 - Difficult to test and debug
- Written by inexperienced programmers

OS Today



OS With Reliability



Objectives

Eliminate **most** downtime caused by drivers

1. Prevent system crashes - **isolation**
2. Keep applications running - **recovery**

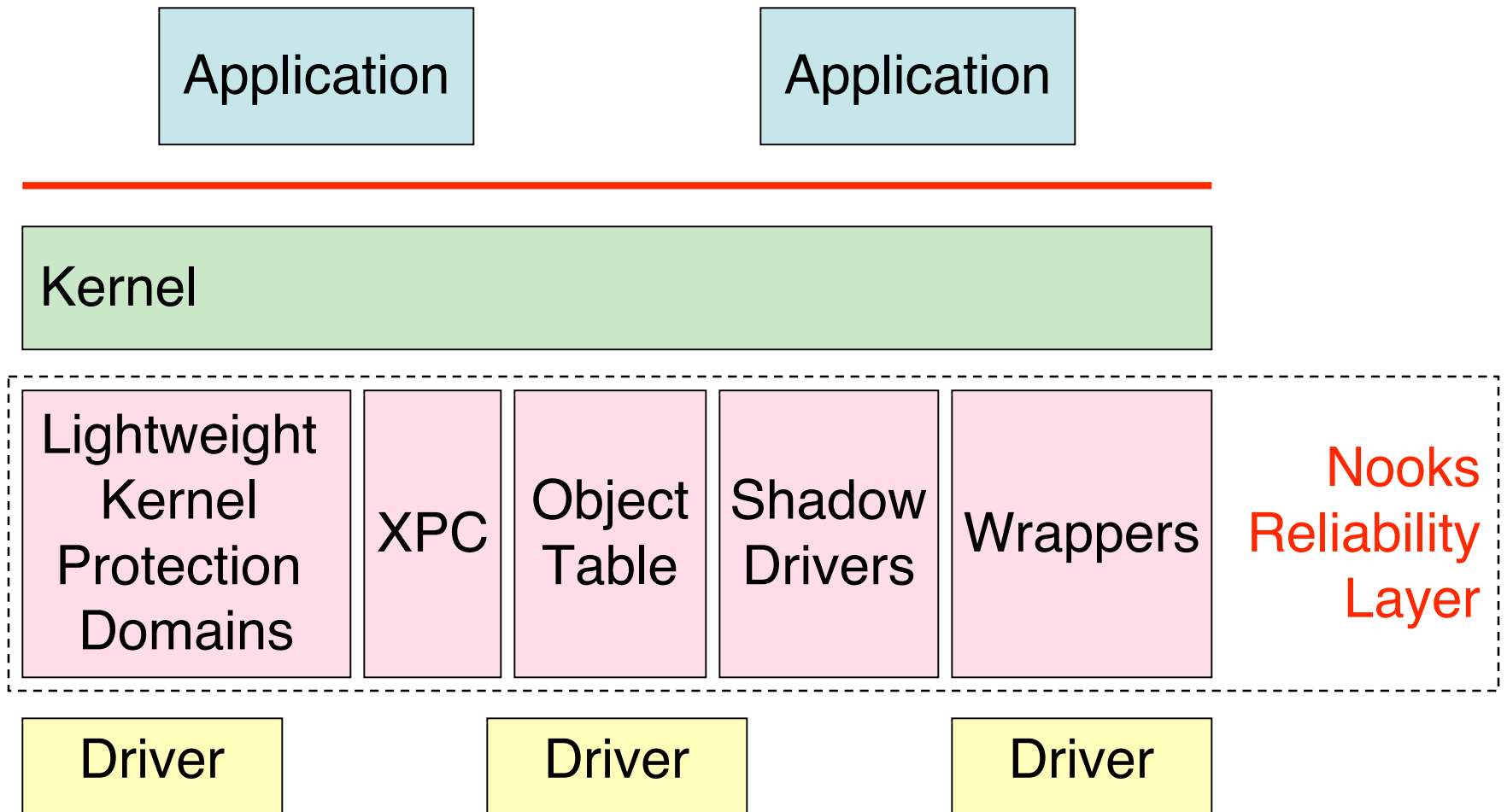
Outline

- Introduction
- Problem
- Design
- Evaluation
- Summary

Design of Nooks

- Standard Linux kernel and drivers
- Plus:
 - Isolation
 - Recovery
- Compatible with existing code

System Architecture



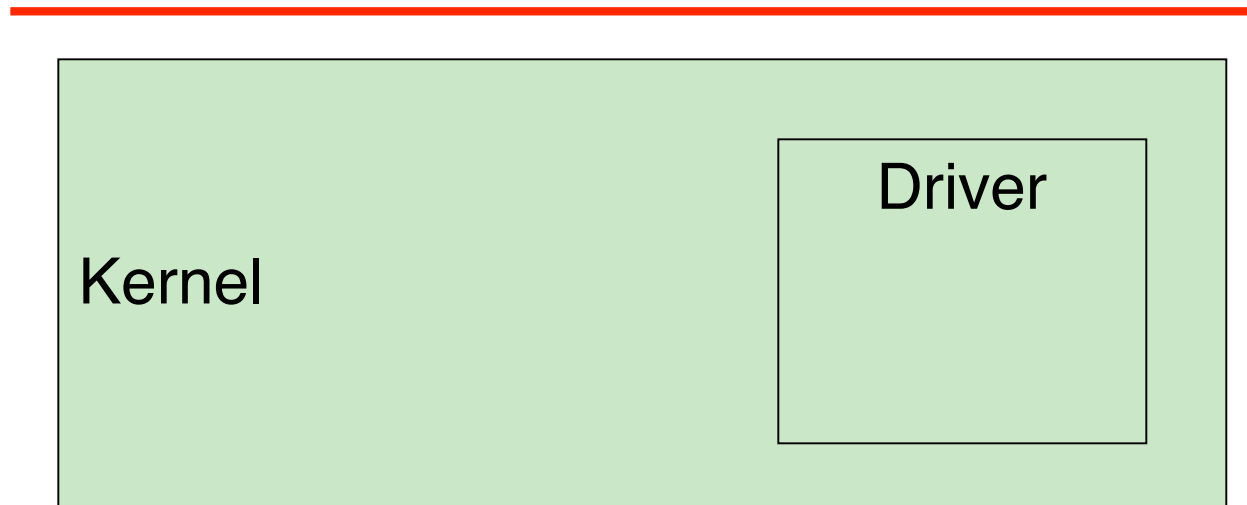
Outline

- Introduction
- Problem
- Design
 - Isolation
 - Recovery
- Evaluation
- Summary

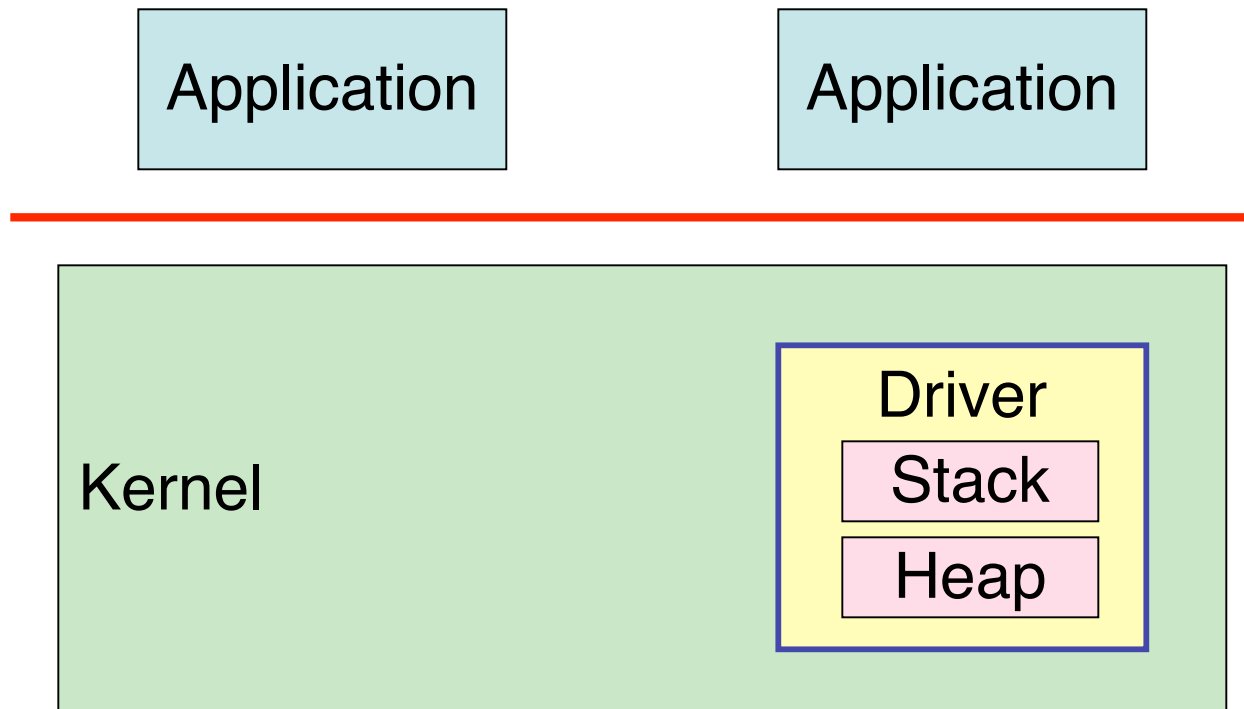
Existing Kernels

Application

Application



Memory Isolation

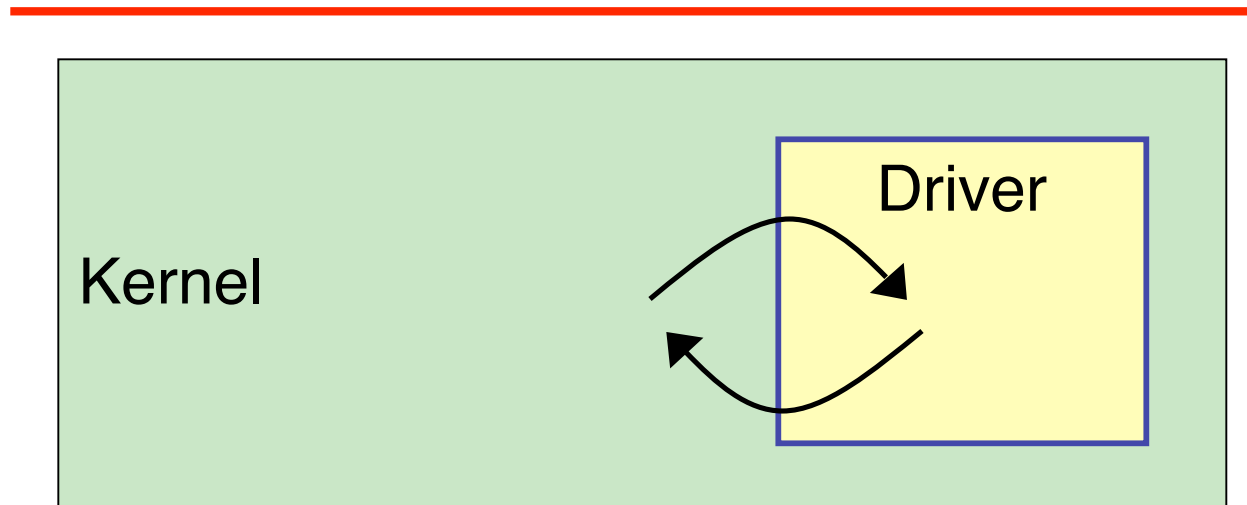


Lightweight Kernel Protection Domains

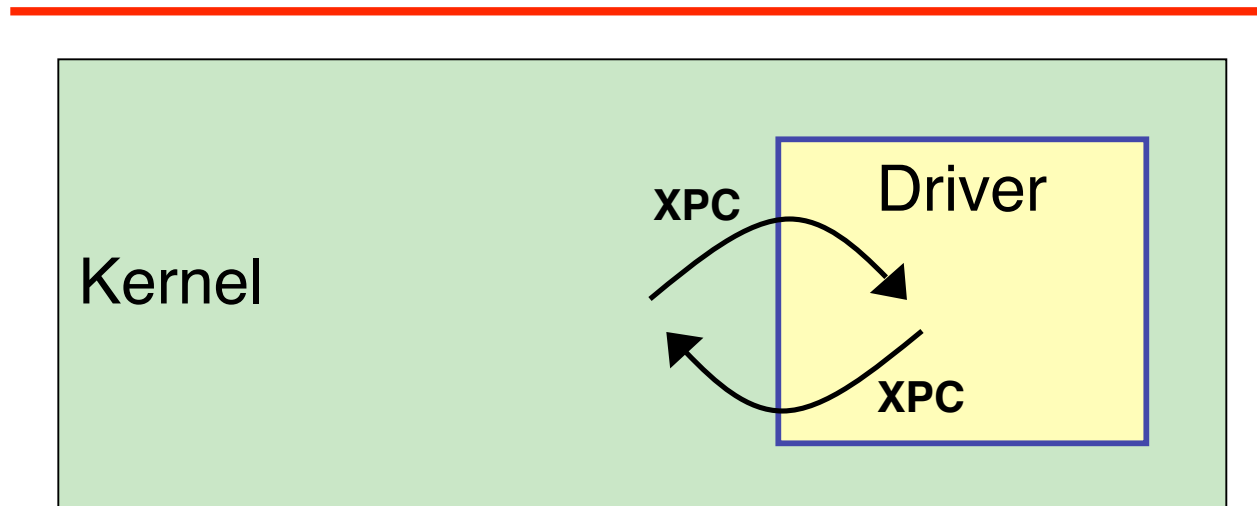
Control Transfer

Application

Application



Control Transfer

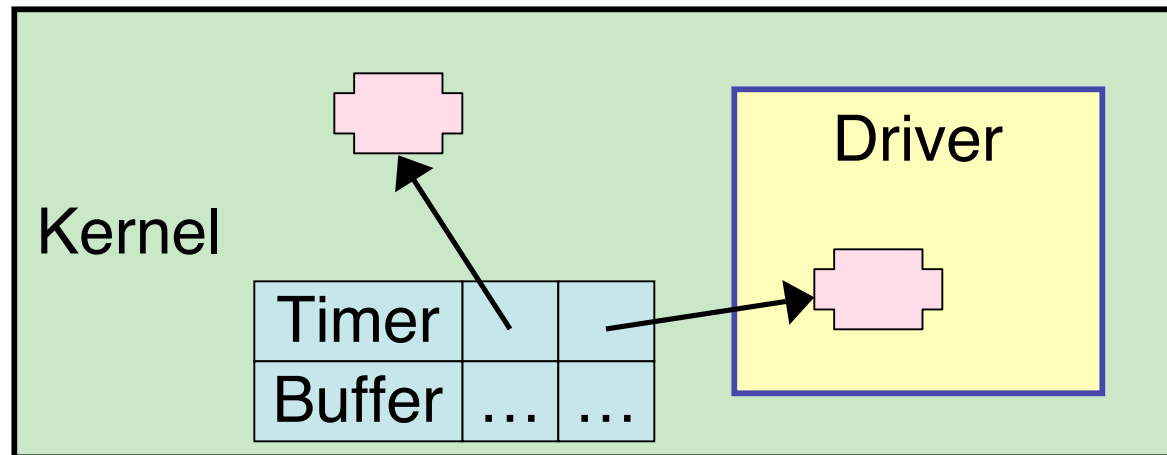


eXtension Procedure Call

Data Access

Application

Application

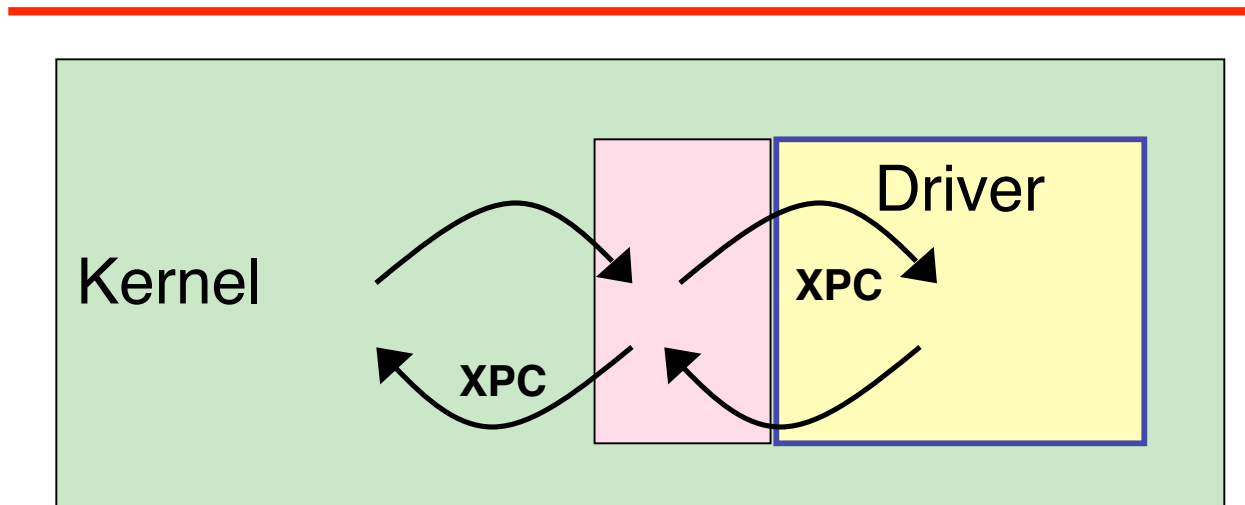


Object Table

Transparency

Application

Application



Wrappers

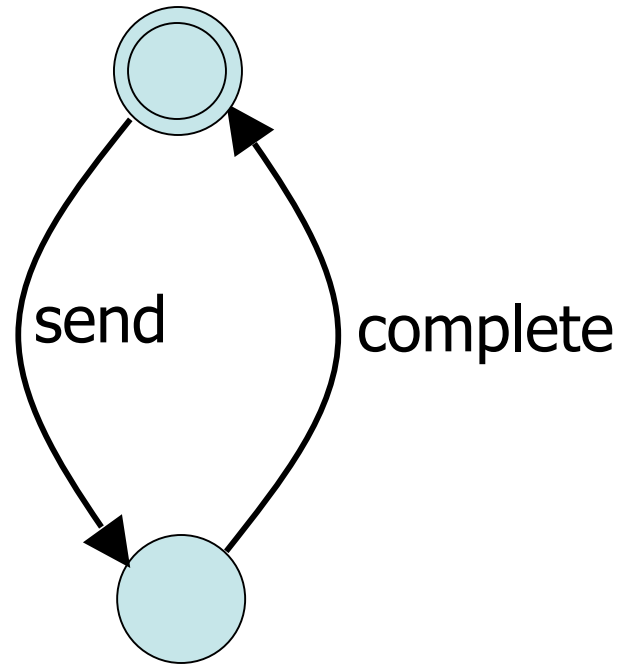
Outline

- Introduction
- Problem
- Design
 - Isolation
 - Recovery
- Evaluation
- Summary

Recovery

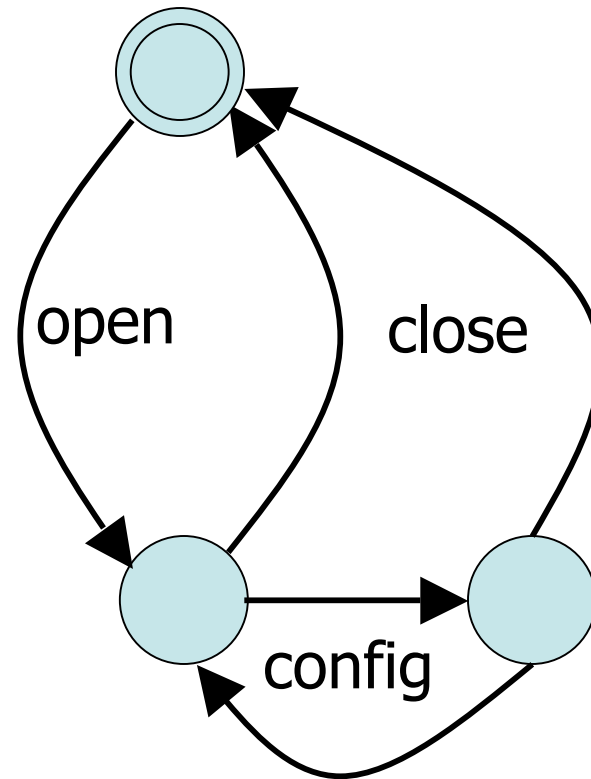
- Goals:
 - Restore driver state so it can process requests as if it had never failed
 - Conceal failure from applications
- Observation:
 - Driver interface specifies how driver responds to requests
- Approach: Model drivers as **state machines**

Drivers as State Machines



Drivers as State Machines

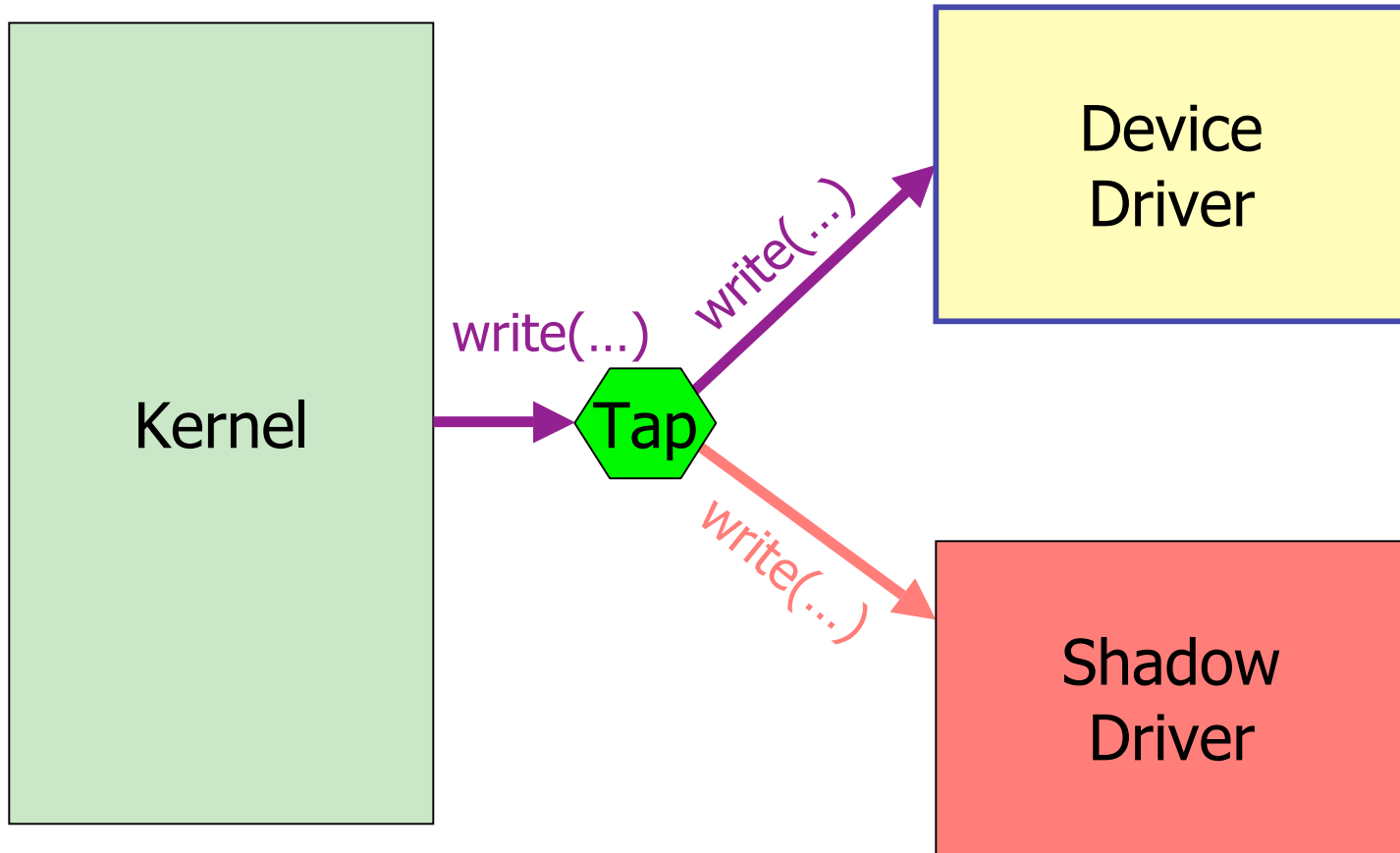
- Recovery:
 - Advance driver from initial state to state at time of crash
 - Reply to requests with valid responses according to driver state



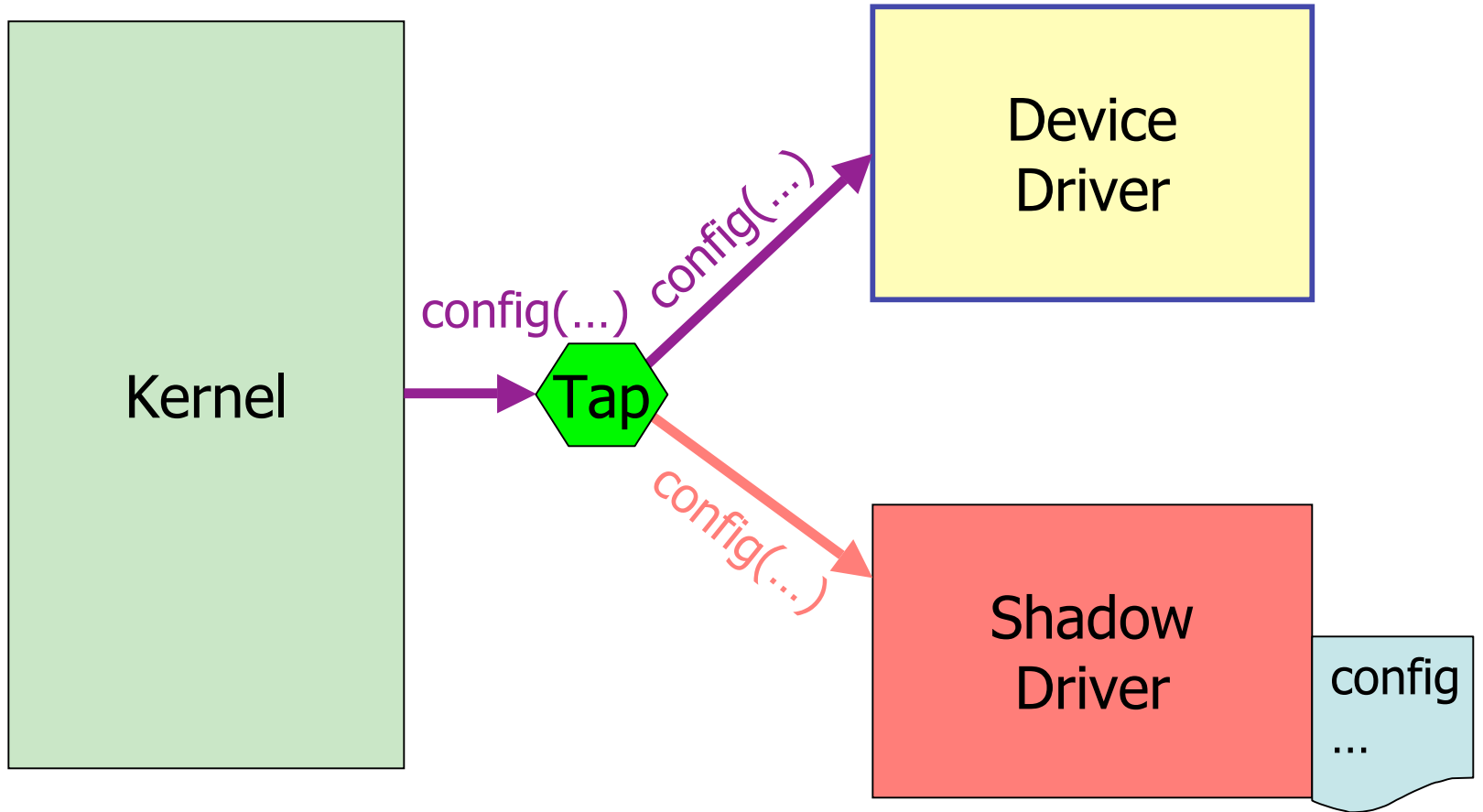
Shadow Drivers

- Generic code that:
 - Normally:
 - Records state-changing inputs
 - On failure:
 - Restarts driver
 - Replays inputs to recover
 - Emulates driver to applications/OS
- **One** shadow driver handles recovery for an **entire class** of drivers

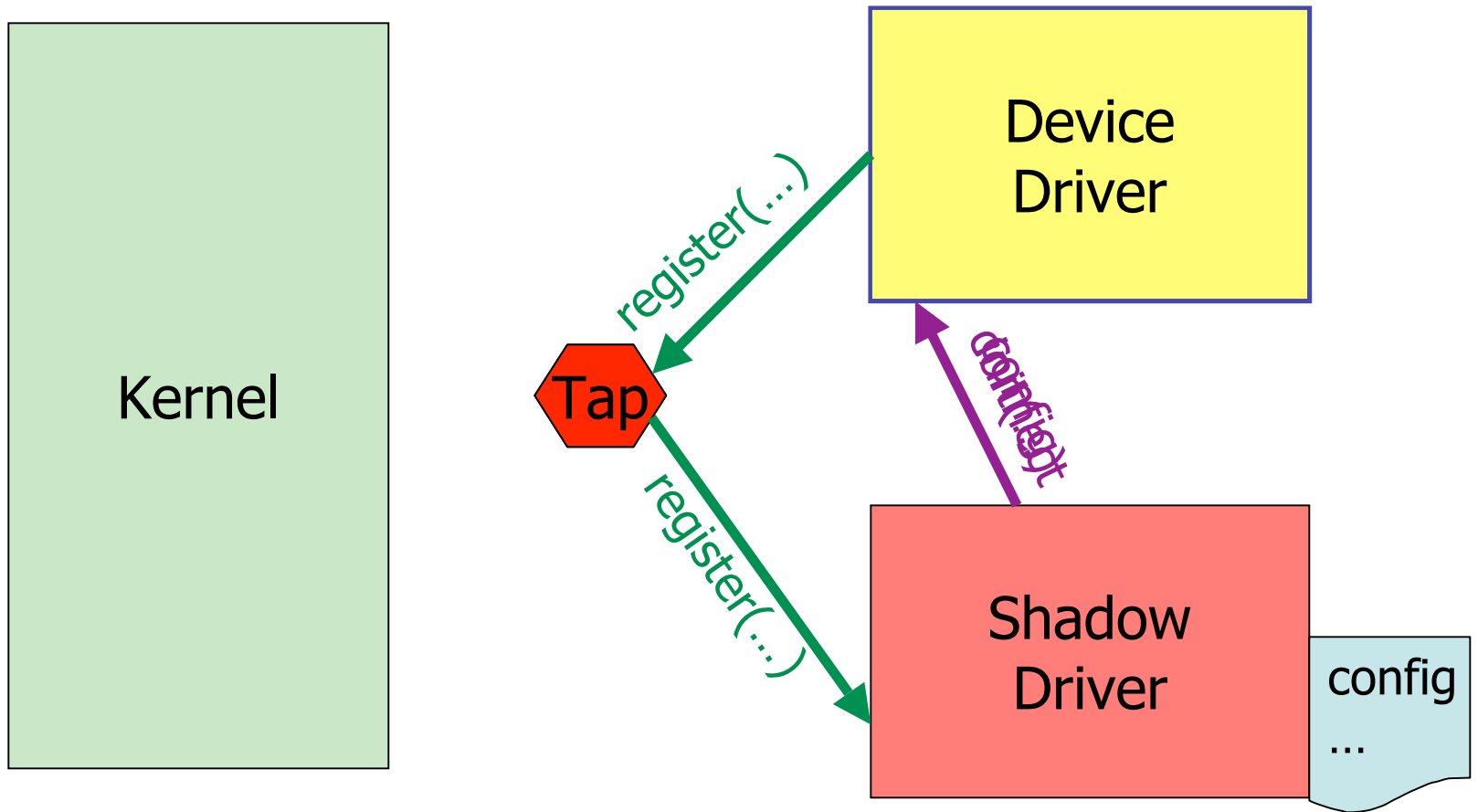
Shadow Driver Overview



Preparing for Recovery



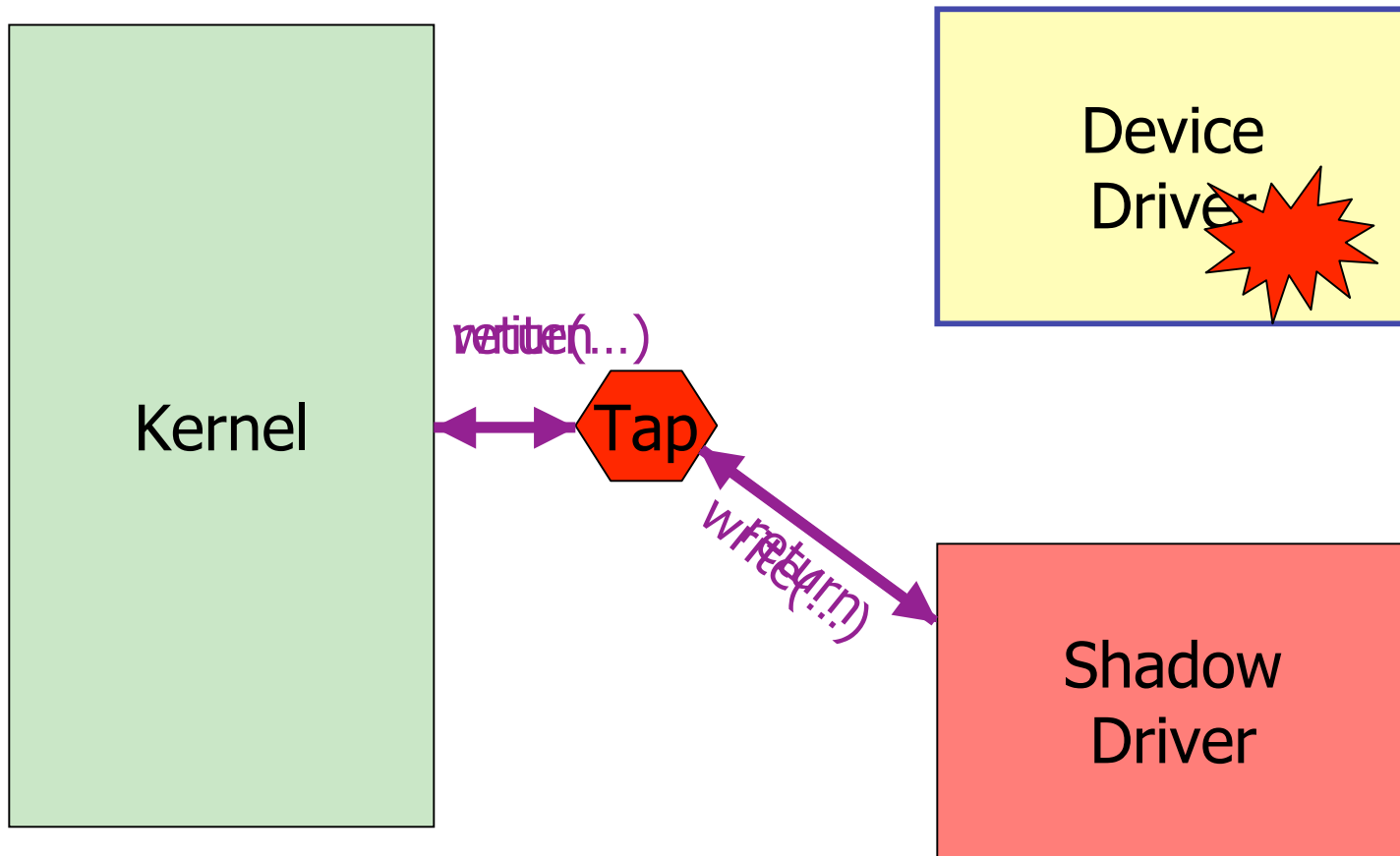
Recovering a Failed Driver



Recovering a Failed Driver

- Summary:
 - Reset driver
 - Reinitialize driver
 - Replay logged requests

Spoofing a Failed Driver



Spoofting a Failed Driver

Shadow acts as driver -- replies to requests with valid **possible** responses

- Applications and OS unaware that driver failed
- No device control

General Strategies:

1. Answer request from log
2. Act busy
3. Block caller
4. Queue request
5. Drop request

Design Summary

- Isolation
 - Lightweight Kernel Protection Domains
 - eXtension Procedure Call (XPC)
 - Object Table
 - Wrappers
- Recovery
 - Shadow Drivers

Outline

- Introduction
- Problem
- Design
- Evaluation
 - Implementation
 - Benefit
 - Cost
- Summary

Drivers Tested



Class	Drivers
Sound	Soundblaster Audigy, Soundblaster 16, Soundblaster Live!, Intel 810 Audio, Ensoniq 1371, Crystal Sound 4232



Network	Intel Pro/1000 Gigabit Ethernet, AMD PCnet32, Intel Pro/100 10/100, 3Com 3c59x 10/100, SMC Etherpower 100
---------	---



IDE Storage	ide-disk, ide-cd
-------------	------------------

Implementation Complexity

- Changes to existing code
 - Kernel: 924 out of 1.1 million lines
 - Device drivers: 0 out of 50,000 lines

Implementation Complexity

- New code
 - Isolation: 23,000 lines
 - Recovery: 3,300 lines

Driver Class	Shadow Driver L.O.C.	1 Device Driver L.O.C.	All Drivers Count	All Drivers L.O.C.
Sound	666	7,381	48	118,981
Network	198	13,577	190	264,500
Storage	321	5,358	8	29,000

Implementation Complexity

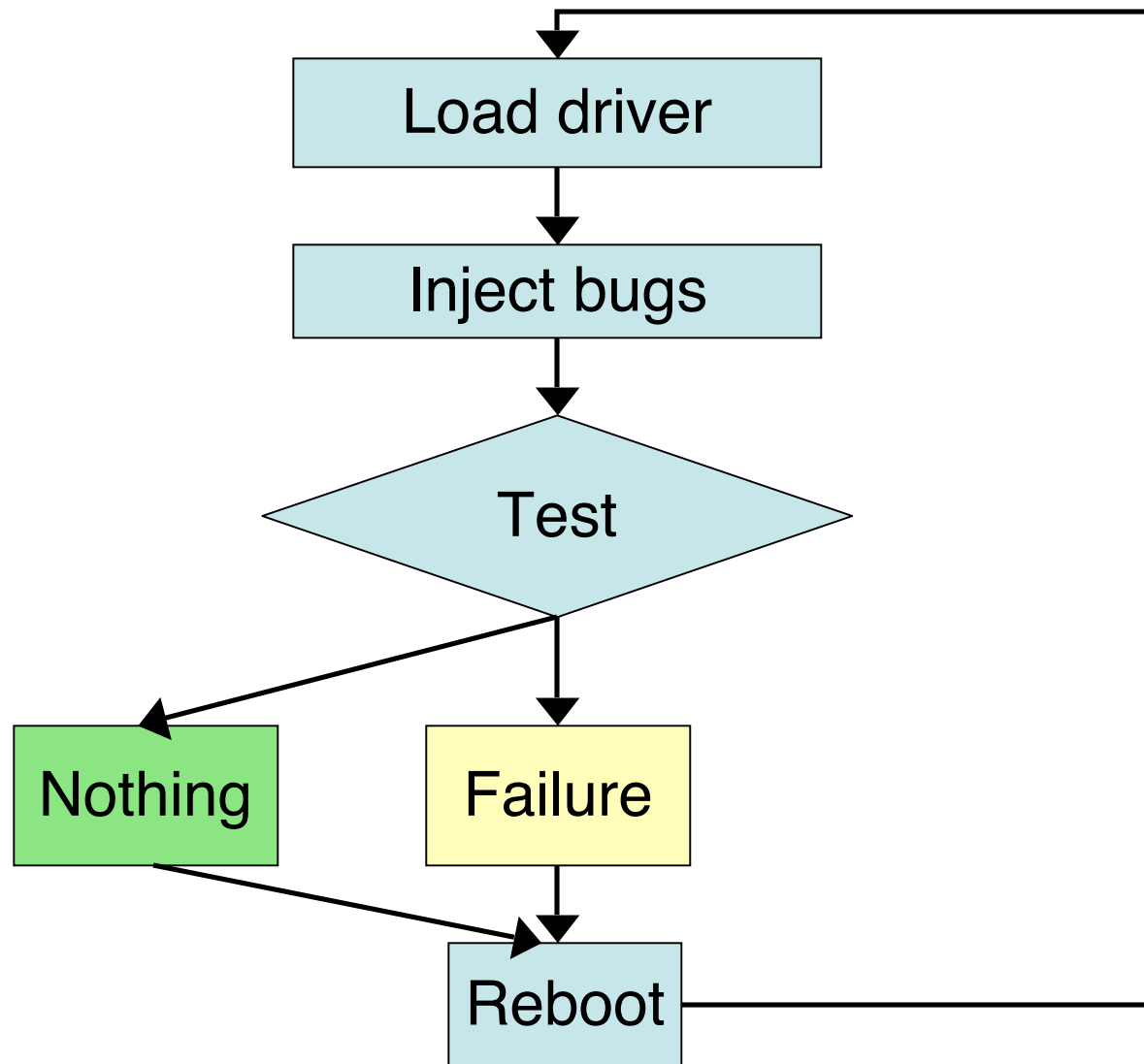
- New code
 - Isolation: 23,000 lines
 - Recovery: 3,300 lines

Driver Class	Shadow Driver L.O.C.	1 Device Driver L.O.C.	All Drivers Count	All Drivers L.O.C.
Sound	666	7,381	48	118,981
Network	198	13,577	190	264,500
Storage	321	5,358	8	29,000

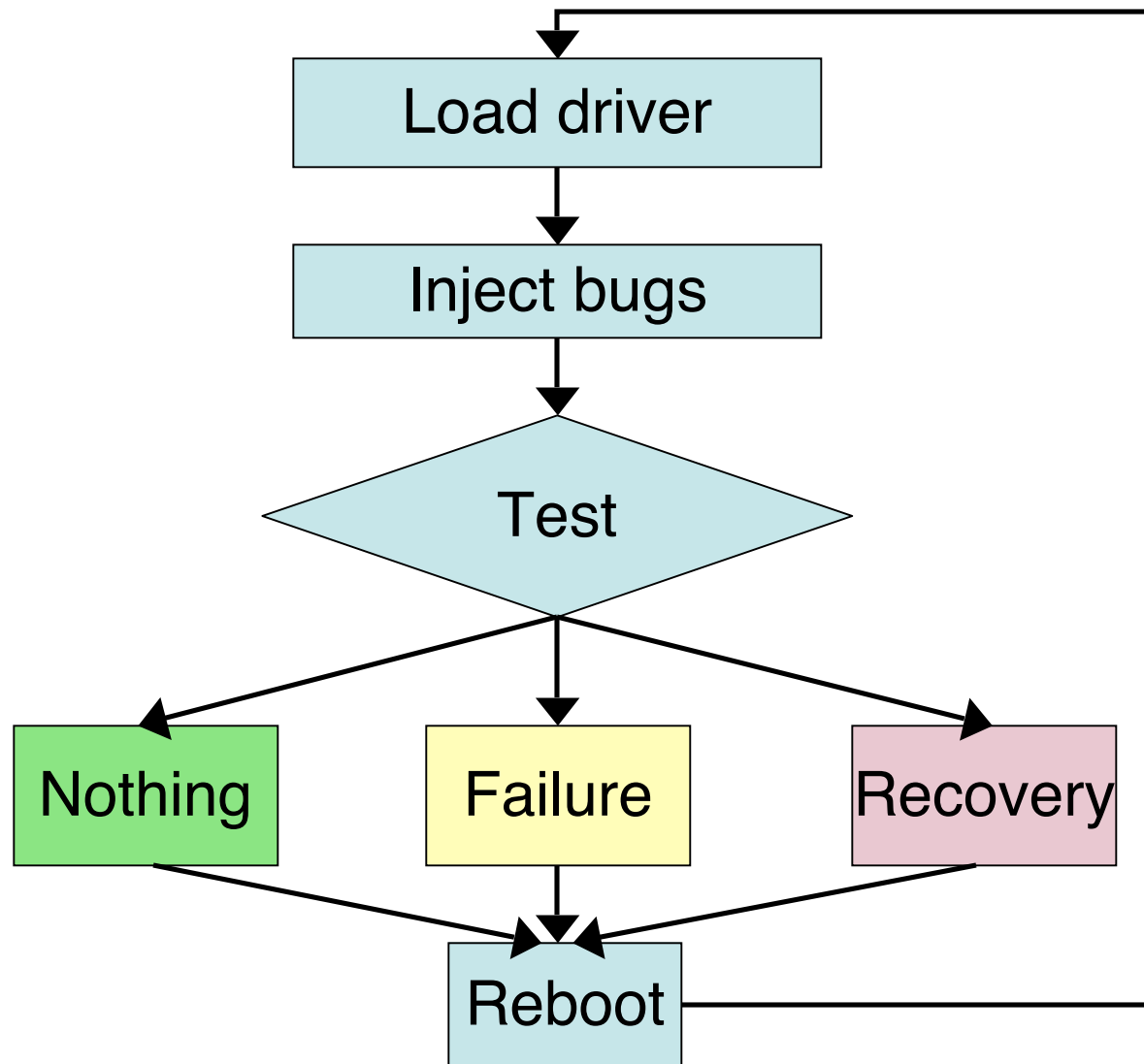
Outline

- Introduction
- Problem
- Design
- Evaluation
 - Implementation
 - Benefit
 - Isolation
 - Recovery
 - Cost
- Summary

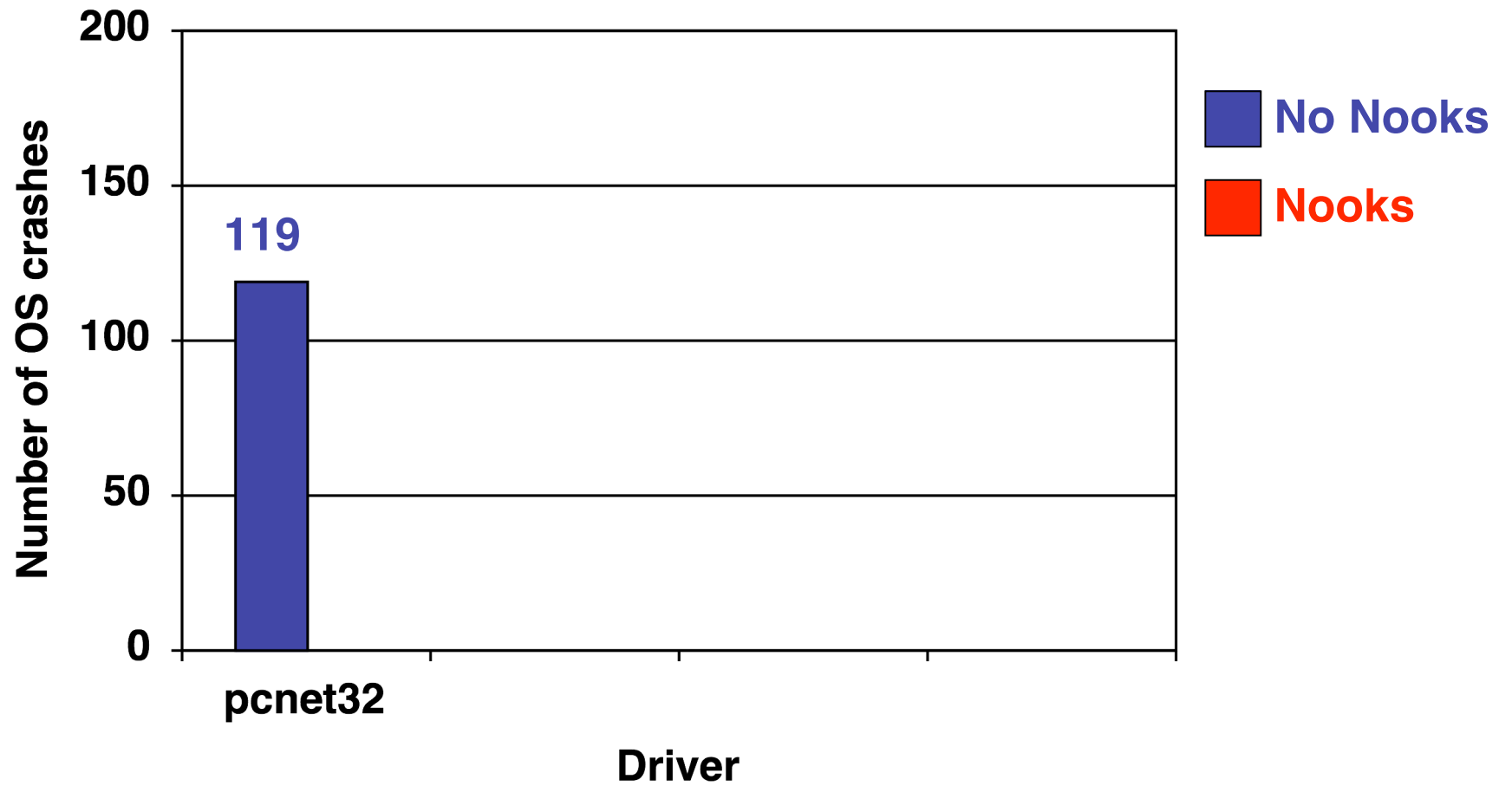
Reliability Test Methodology



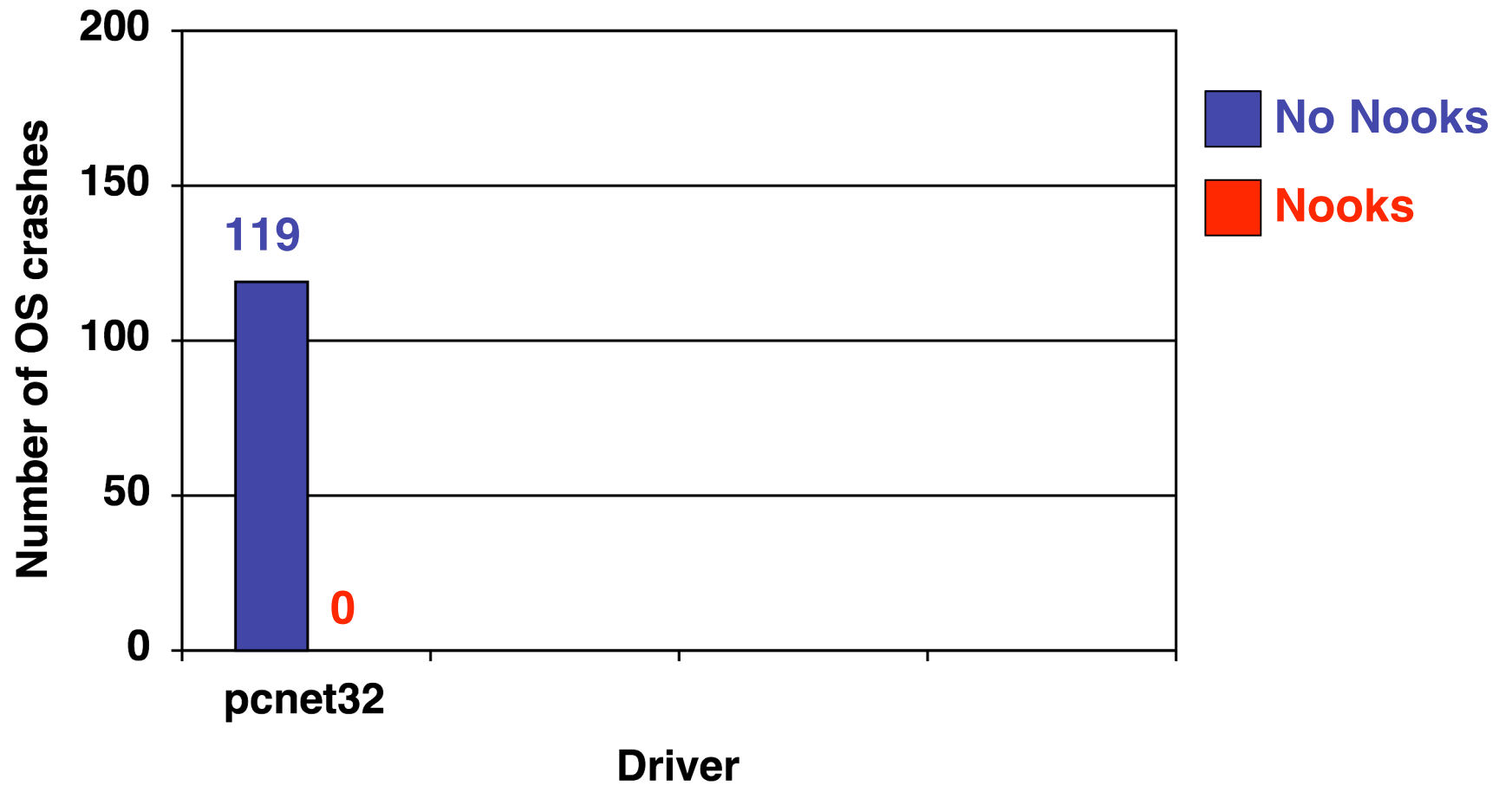
Reliability Test Methodology



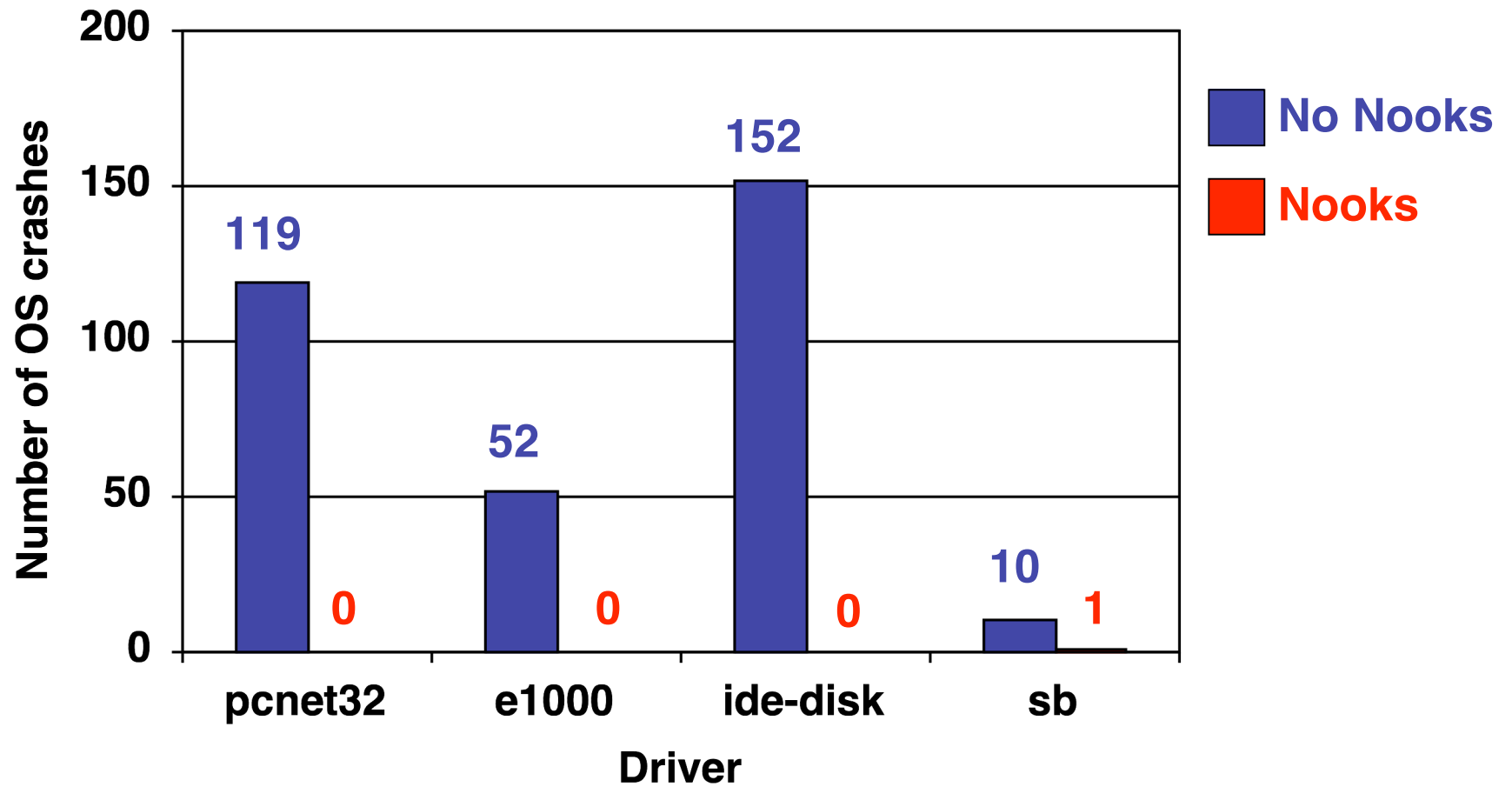
Isolation Works



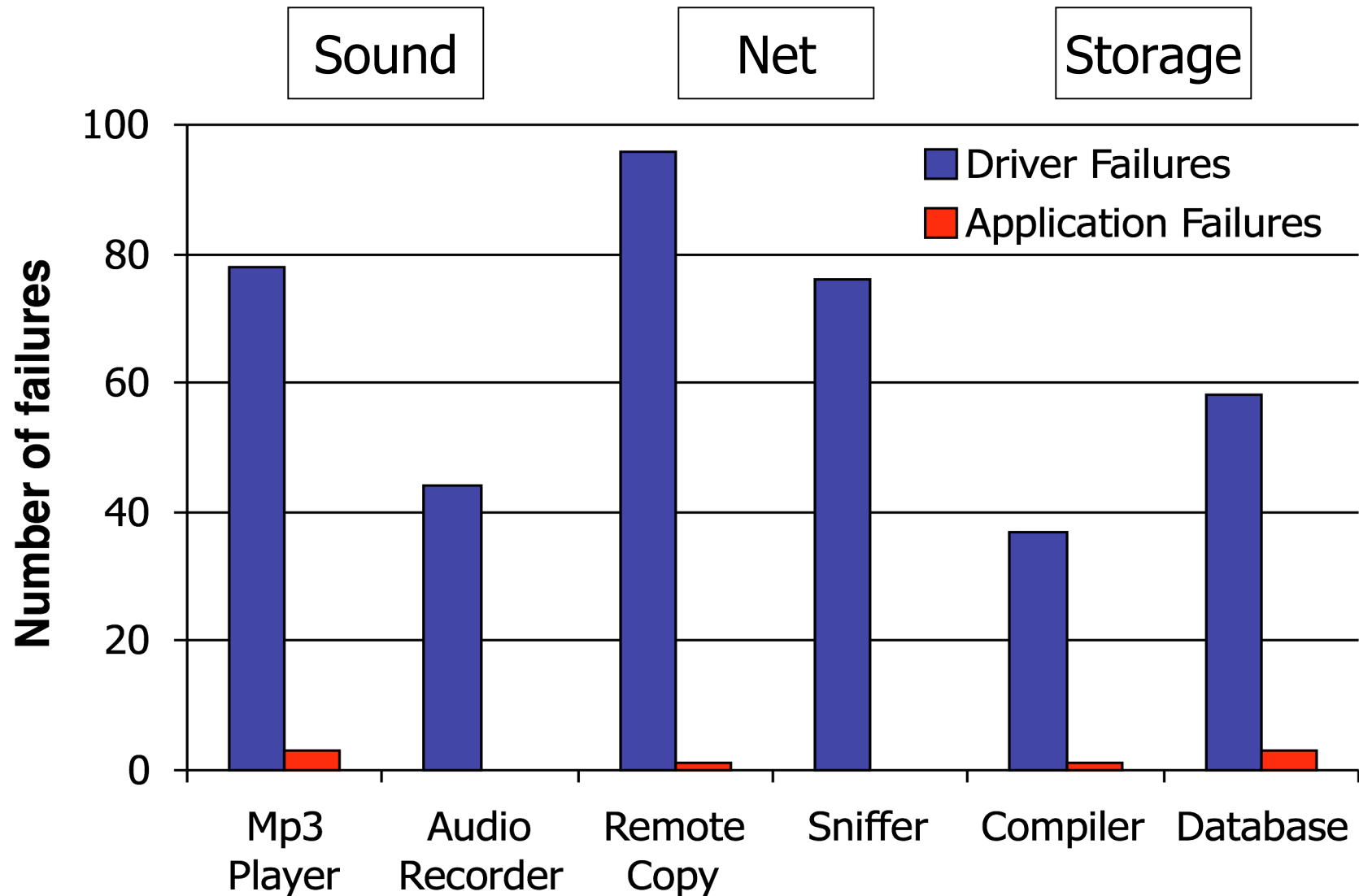
Isolation Works



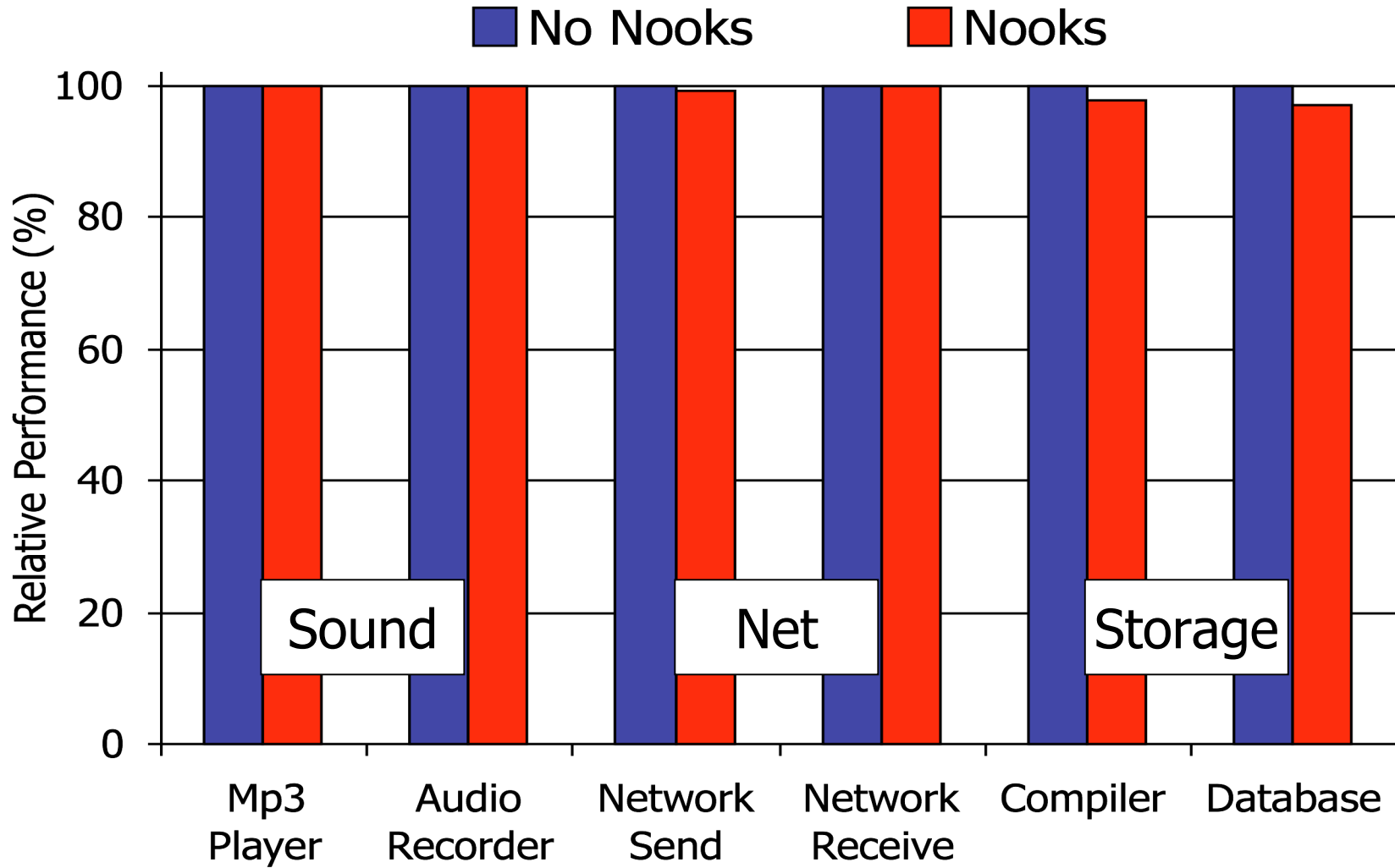
Isolation Works



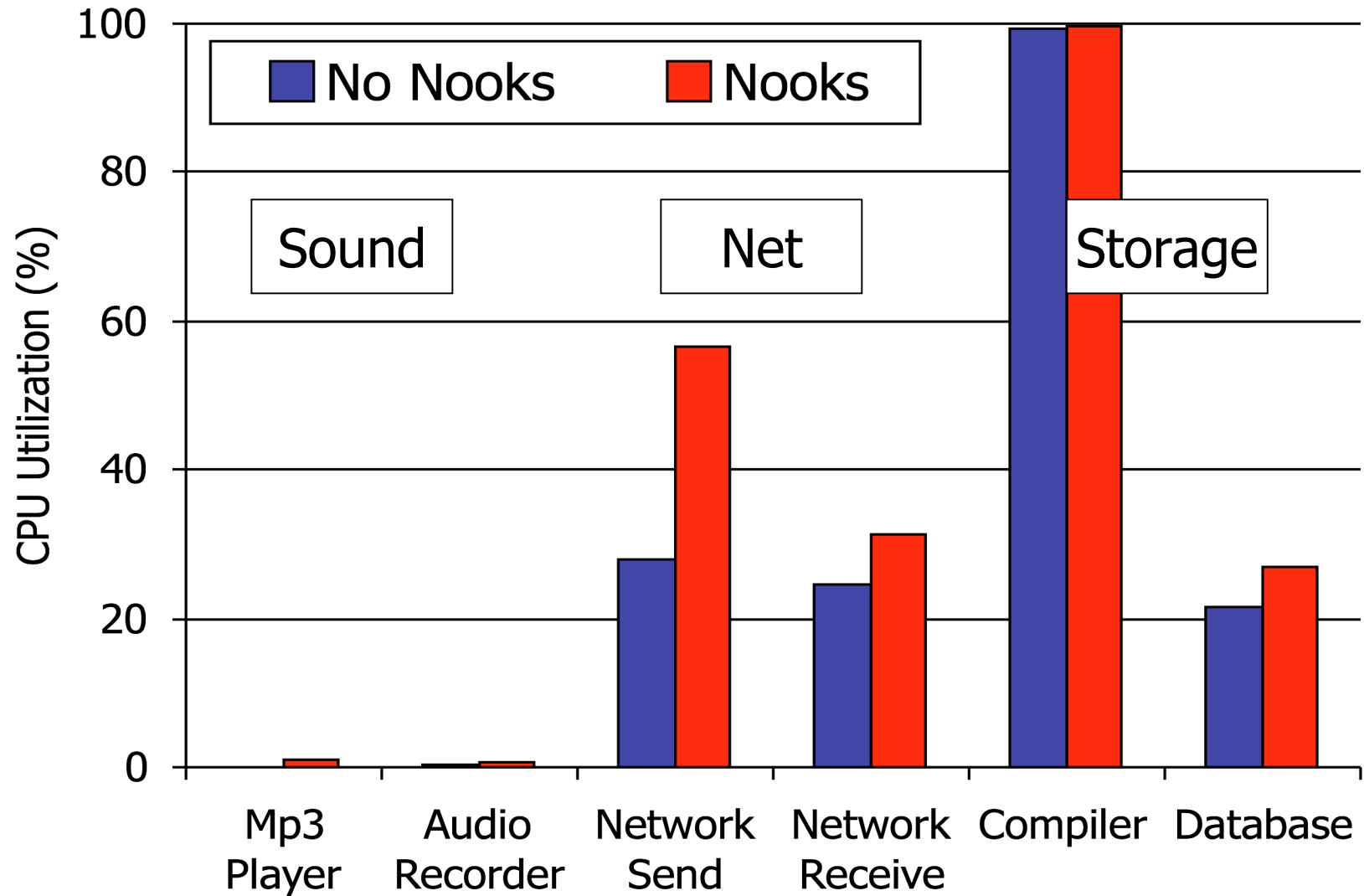
Recovery Works



Relative Performance



CPU Usage



Future Work

- Application to virtual machines
 - Apply shadow drivers to Xen
- Fixing drivers
 - Reducing hardware-induced driver failures
 - New driver architectures
 - Testing drivers w/o devices
 - Writing drivers in Python

Conclusion

- Nooks is an architecture and a set of components and techniques for improving system reliability that is:
 - Highly effective at preventing crashes
 - Compatible with existing code
 - Low performance overhead
 - Low implementation cost

Questions?