Granary: Comprehensive Kernel Module Instrumentation

Peter Goodman Akshay Kumar Angela Demke Brown Ashvin Goel

University of Toronto

Modules are hard to analyse

Debugging, testing, and securing modules is challenging
- Tight interaction with the kernel
- Sometimes distributed as binaries
- Asynchronous and concurrent execution

A module analyser should...

- Comprehensively instrument all binary modules
- Impose no performance overhead on non-module kernel code
- Require no changes to existing or future modules
- Require minimal changes to the kernel

Approach: mixed-mode execution

1

Motivation: Comprehensive module instrumentation with no overhead to kernel code.

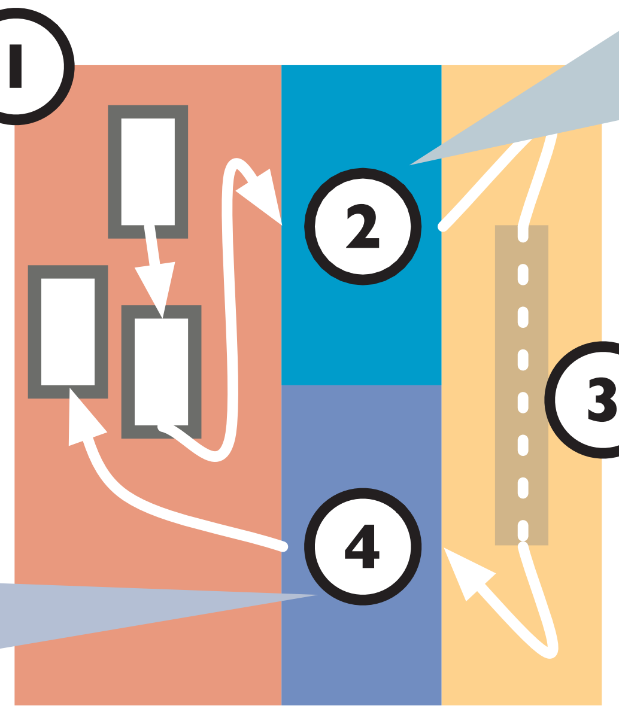
Key Idea: Use dynamic binary translation to control and instrument all module code; don't instrument kernel code.

Challenges: When/how to take and relinquish control.

2

Exit Instrumentation via Wrapped Functions
Granary relinquishes control when an instrumented module calls a kernel function. Before doing so, Granary needs to ensure that it can regain control when module code is invoked.

- Finds kernel interface functions dynamically; recursively wraps argument data structures
- The wrappers change pointers to module functions passed to the kernel into pointers to shadow module functions



4

Enter Instrumentation via Shadow Modules
Granary regains control when the kernel returns to the module or invokes a shadow module pointer.

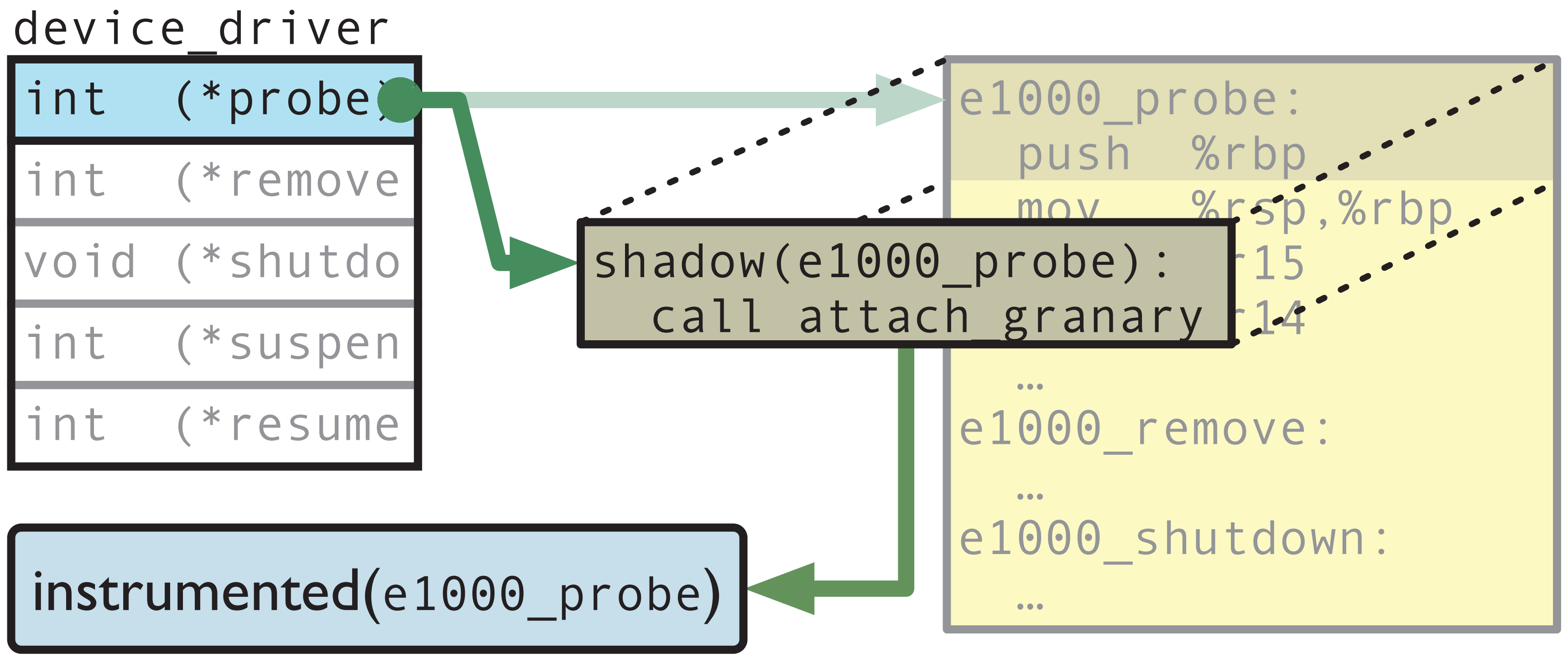
3

Kernel Code Executes Natively
All non-module kernel code, including interrupt and exception handlers, runs without instrumentation.

Wrapping

Problem
- Granary does not control the execution of kernel code
- Modules share function pointers with the kernel
- Granary must gain control when the kernel invokes any module function pointer

Solution
- All arguments to kernel functions are wrapped
- Wrapping changes function pointers in arguments into shadow function pointers so that Granary regains control



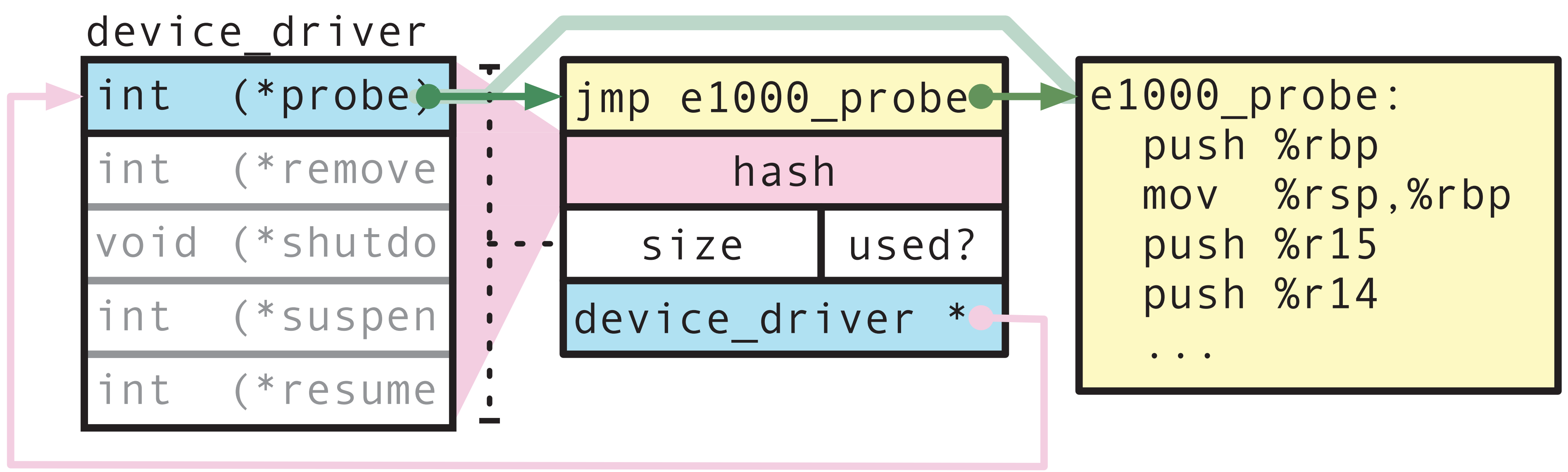
```
device_driver
int   (*probe)
int   (*remove
void  (*shutdo
int   (*suspen
int   (*resume
```

```
shadow(e1000_probe):
  call attach_granary
```

```
e1000_probe:
  push  %rbp
  mov   %rsp,%rbp
  ...15
  ...14
  …
e1000_remove:
  …
e1000_shutdown:
  …
```

instrumented(e1000_probe)

Avoiding redundant argument wrapping



```
device_driver
int   (*probe)
int   (*remove
void  (*shutdo
int   (*suspen
int   (*resume
```

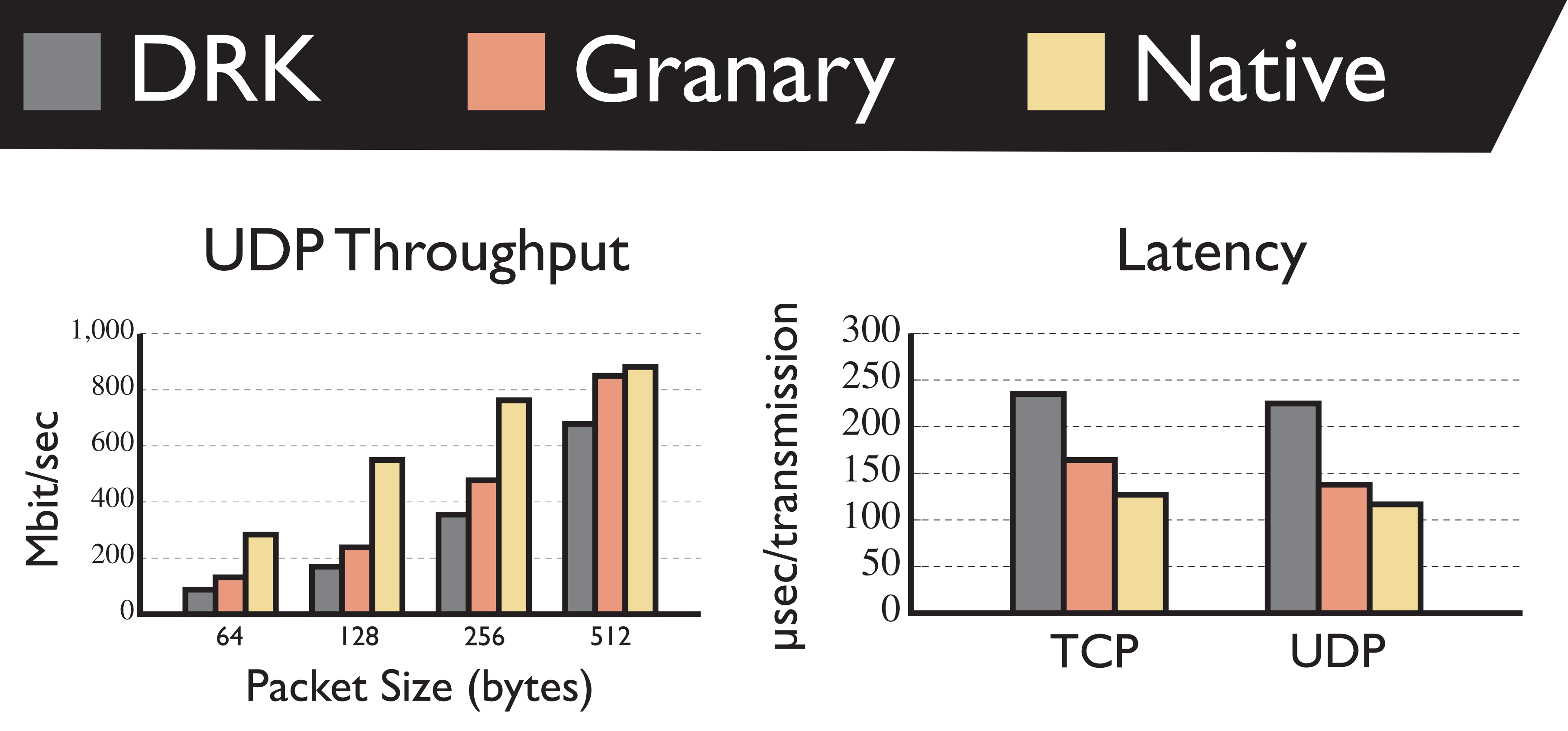
```
jmp e1000_probe
hash
size   used?
device_driver *
```

```
e1000_probe:
  push %rbp
  mov  %rsp,%rbp
  push %r15
  push %r14
  ...
```

Problem
- Deeply linked/nested data structures passed as arguments can contain function pointers
- Wrapping these arguments is expensive

Solution
- Wrap an argument only if the value it points to has changed
- Store a hash of the data structure passed as an argument to check if it has changed
- Override a function pointer in the argument to store a hash

Performance benchmarks



We benchmarked Granary against:
- **Native**: Uninstrumented **e1000e** network driver
- **DRK**: DynamoRIO Kernel-instrumented Linux kernel and the **e1000e** network driver

If the CPU is fully utilized then Granary incurs a 10% to 50% decrease in UDP throughput. If the CPU is not fully utilized then Granary has no effect on TCP throughput.

With a message size of one byte, network latency with Granary increases by at most 20%.