

XXX

Tolerating Malicious Drivers in Linux

Silas Boyd-Wickizer and Nickolai Zeldovich

How could a device driver be malicious?

Today's device drivers are highly privileged

Write kernel memory, allocate memory, ...

Drivers are complex; developers write buggy code

Result: Attackers exploit vulnerabilities

How could a device driver be malicious?

Today's device drivers are highly privileged

Write kernel memory, allocate memory, ...

Drivers are complex; developers write buggy code



The screenshot shows a ZDNet article page. At the top left is the ZDNet logo. Below it is a navigation bar with links for 'News & Blogs', 'Reviews', 'Downloads', and 'White Papers'. A secondary navigation bar includes 'US Edition', 'Companies', 'Hardware', 'Software', 'Mobile', 'Security', and 'Re'. The main content area features the article title 'Surge of killer device drivers leave no OS safe' in a large, bold font. Above the title is the author's name 'George Ou' and the publication 'Real World IT'. To the right of the author information are links for 'Mobile', 'RSS', and 'Email Alerts'. Below the title is a social media sharing bar with icons for '82 Comments', 'Share', 'Print', 'Facebook', 'Twitter', 'Recommend', and '0 Votes'. The breadcrumb trail at the bottom of the article reads 'Home / News & Blogs / Real World IT'.

vulnerabilities

How could a device driver be malicious?

Today's device drivers are highly privileged

Write kernel memory, allocate memory

Drivers are C code

The screenshot shows the ZDNet website interface. At the top left is the ZDNet logo. Below it, there are navigation links for 'News & Blogs' and 'Companies'. A red banner indicates 'US Edition'. The main content area features a large headline: 'Surge of killer device drivers leave no OS safe'. Below the headline, there are social media sharing options for comments, share, and print. The article title 'Real World IT' by George Ou is also visible.

The image displays a stack of four screenshots from the National Vulnerability Database (NVD) showing details for different CVEs. The top-most screenshot is for CVE-2009-4537. The second screenshot is for CVE-2010-1085. The third screenshot is for CVE-2010-0410. The bottom-most and largest screenshot is for CVE-2010-1086, which is currently under review. This entry includes a detailed description: 'The ULE decapsulation functionality in drivers/media/dvb/dvb-core/dvb_net.c in dvb-core in Linux kernel 2.6.33 and earlier allows attackers to cause a denial of service (infinite loop) via a crafted MPEG2-TS frame, related to an invalid Payload Pointer ULE.' It also lists several references, including MLIST entries, CONFIRM entries from git.kernel.org and bugzilla.redhat.com, and advisory entries from DEBIAN (DSA-2053), REDHAT (RHSAs-2010-0398 and -0394), and SUSE (SUSE-SA-2010-019).

Current approach

User-space drivers in μ kernels (Minix, L4, ...)

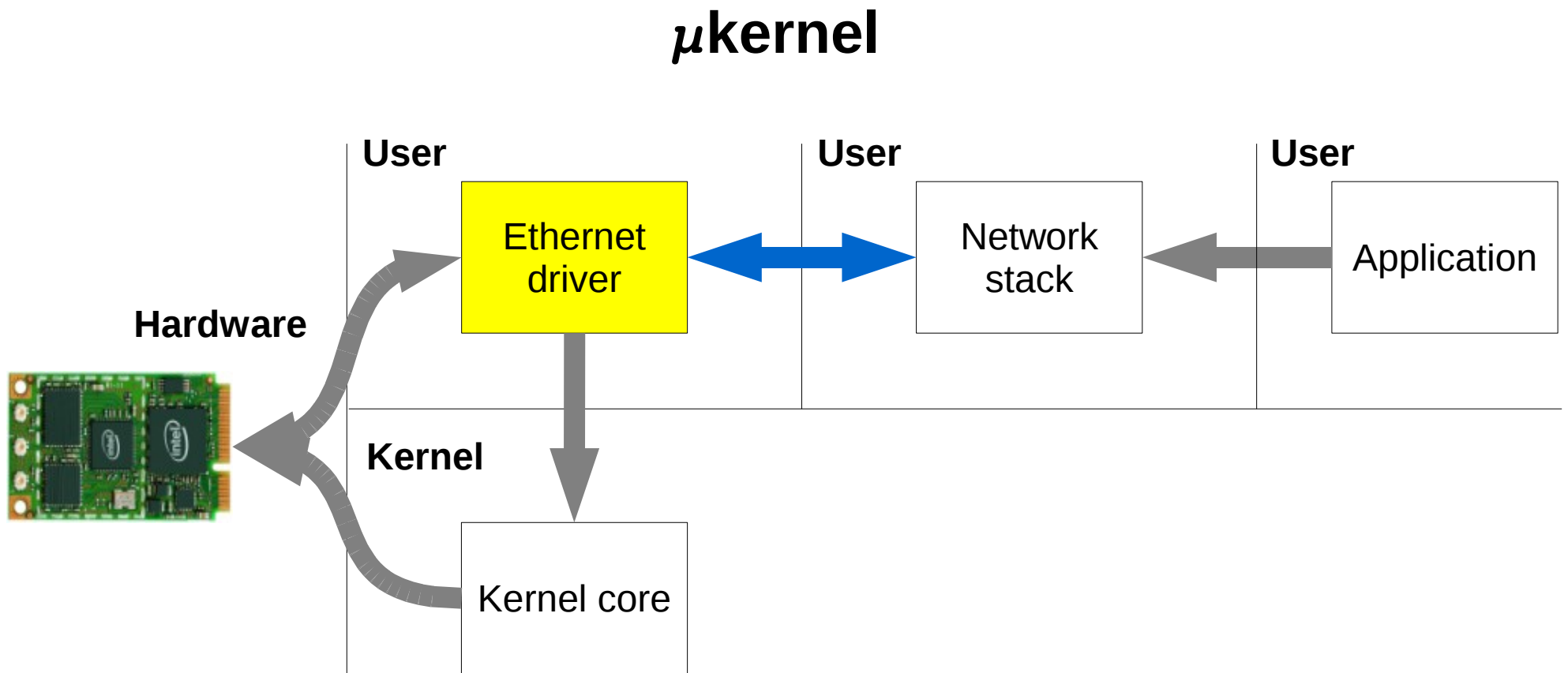
Write device driver in new language (Termite)

Handle common faults (Nooks, microdrivers, ...)

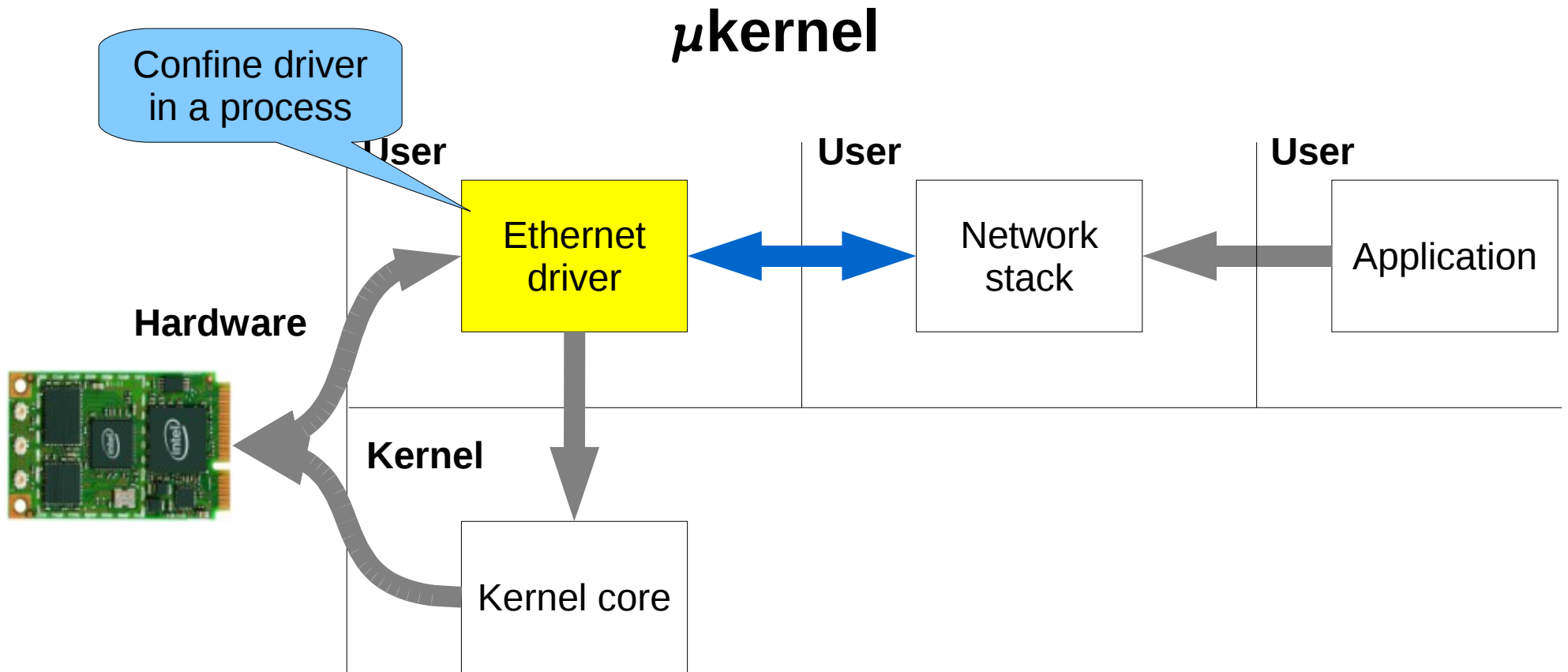
Goal

Secure, efficient, & unmodified
drivers on Linux

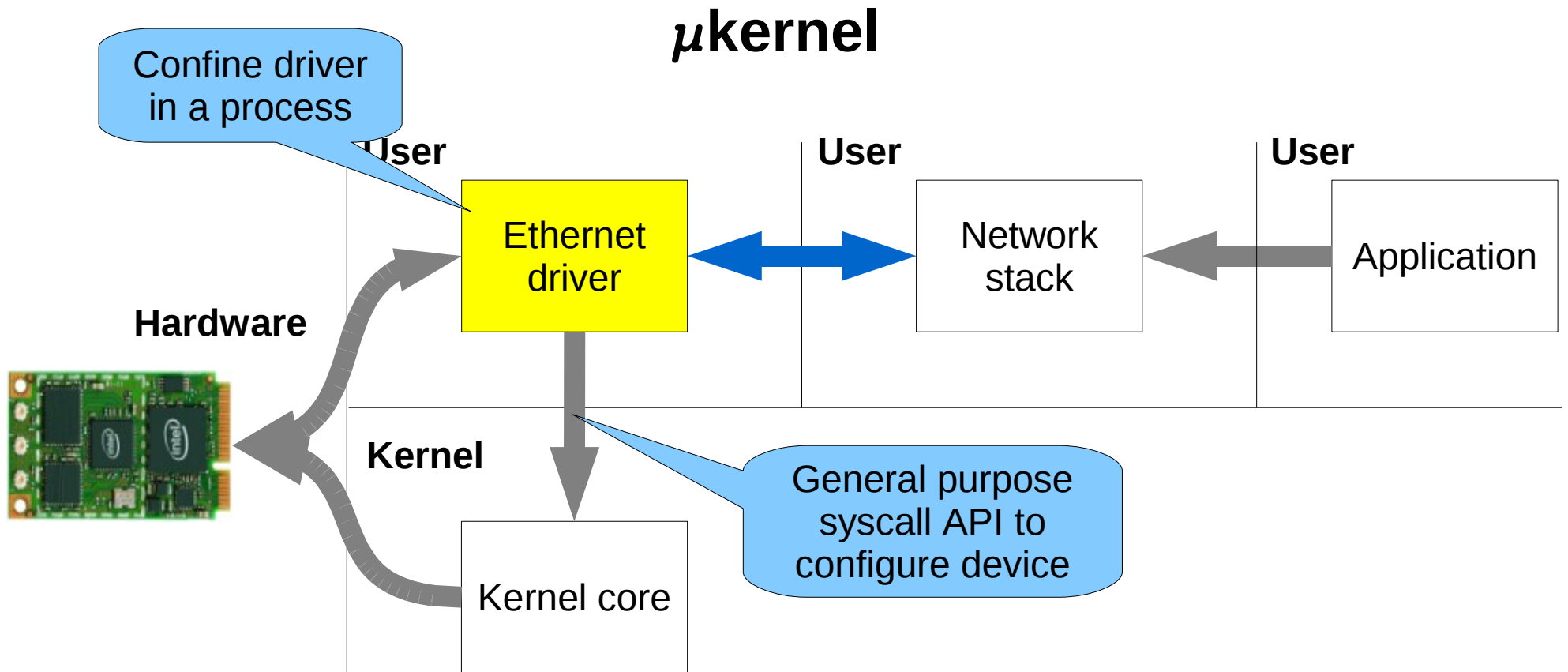
Previous user-space drivers



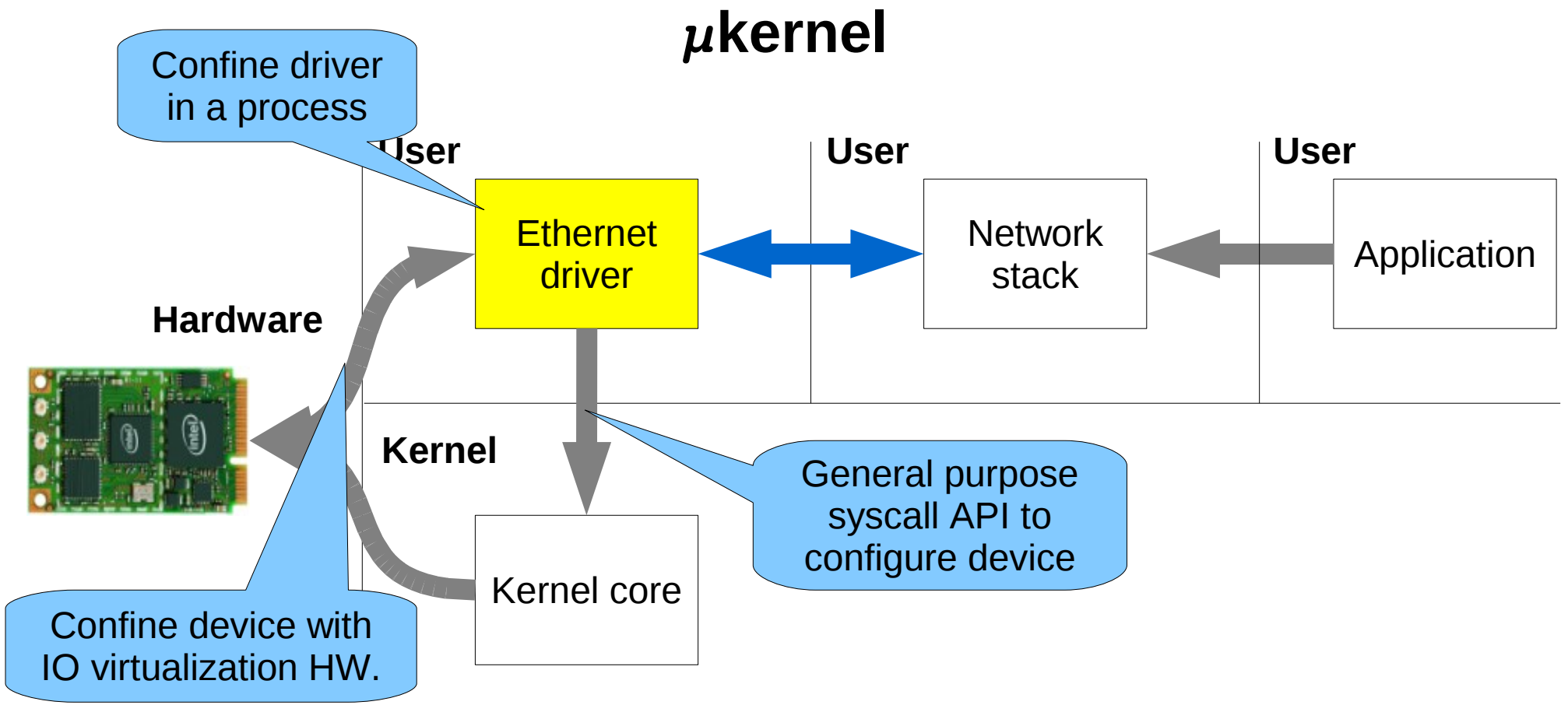
Previous user-space drivers



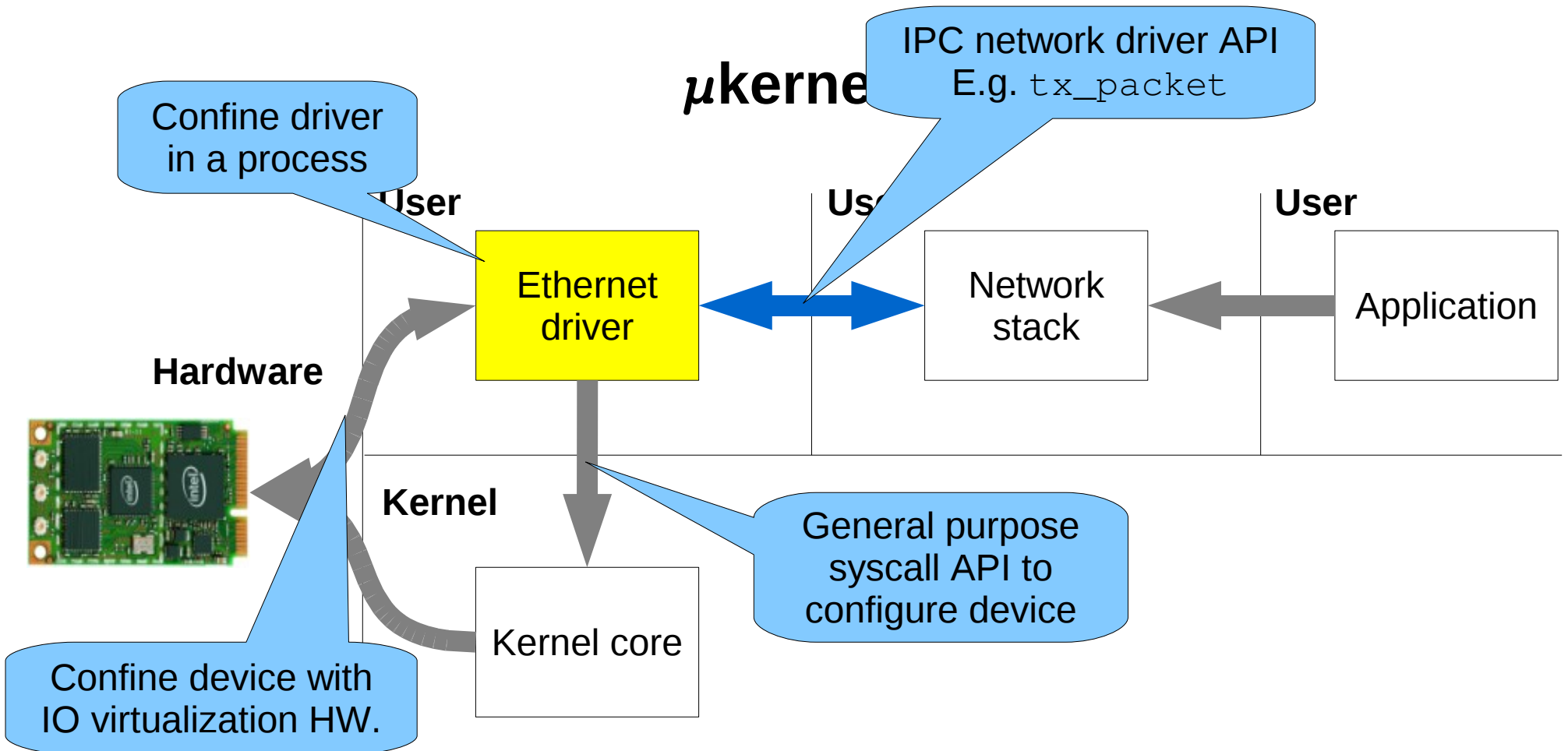
Previous user-space drivers



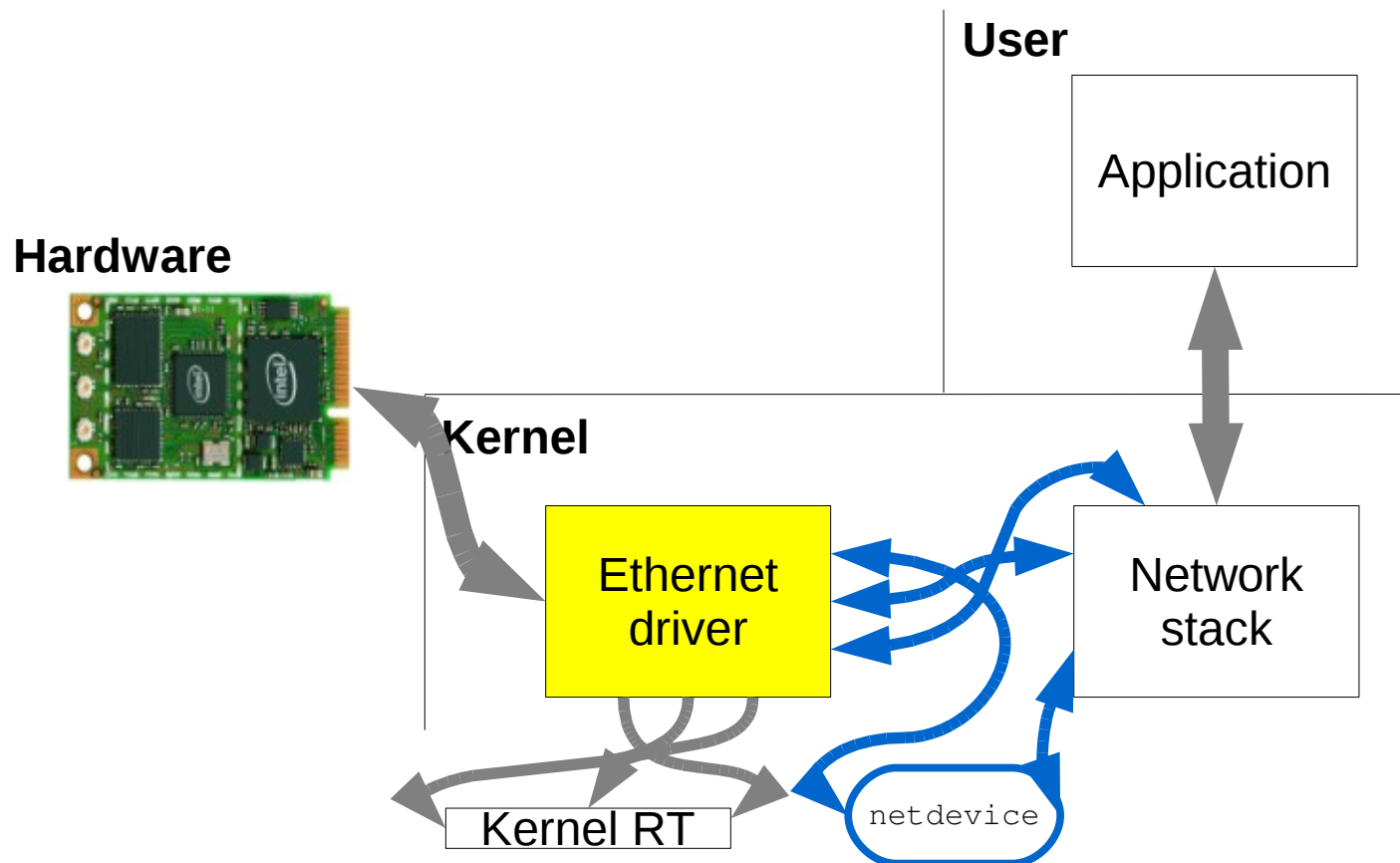
Previous user-space drivers



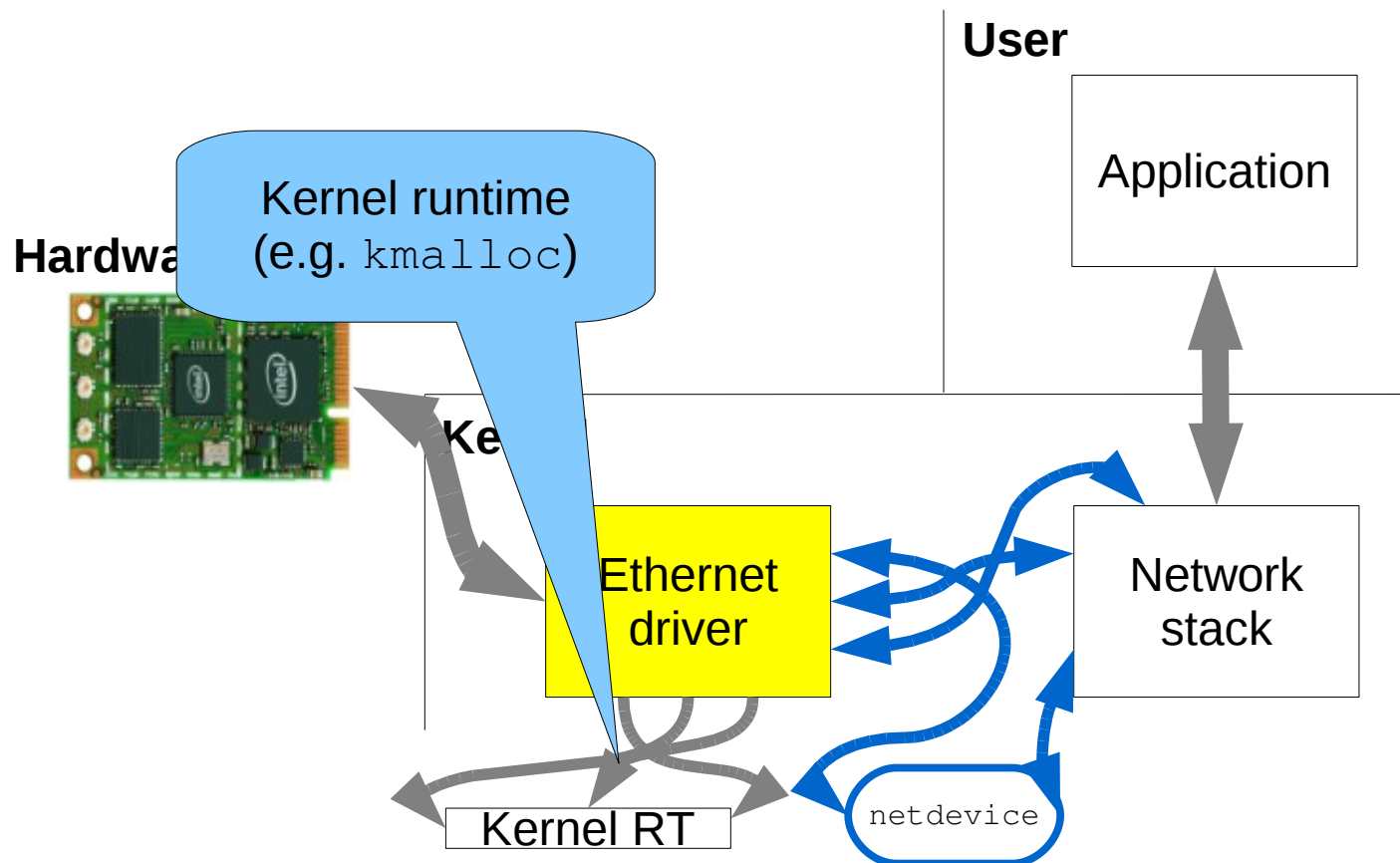
Previous user-space drivers



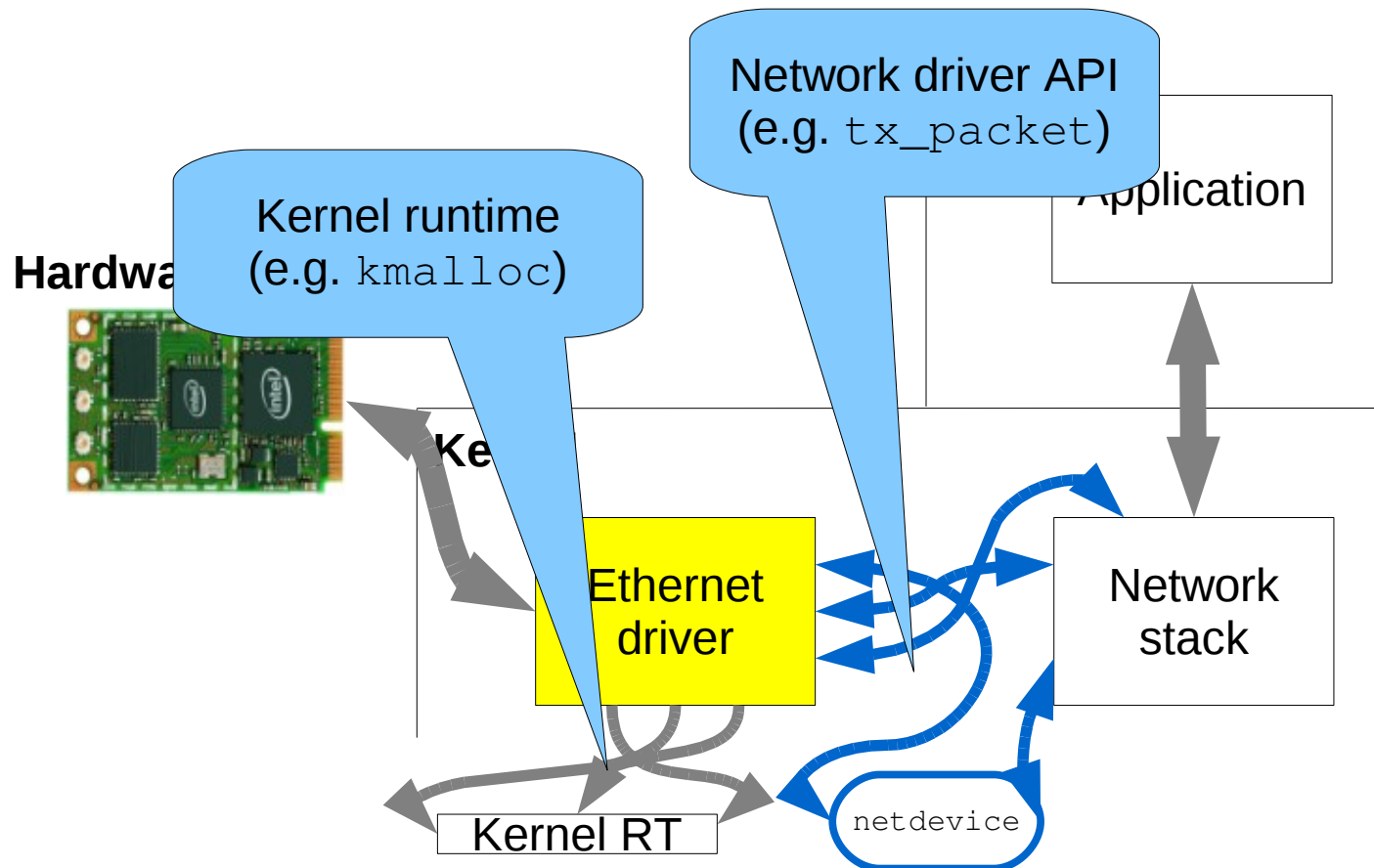
Current Linux driver architecture



Current Linux driver architecture

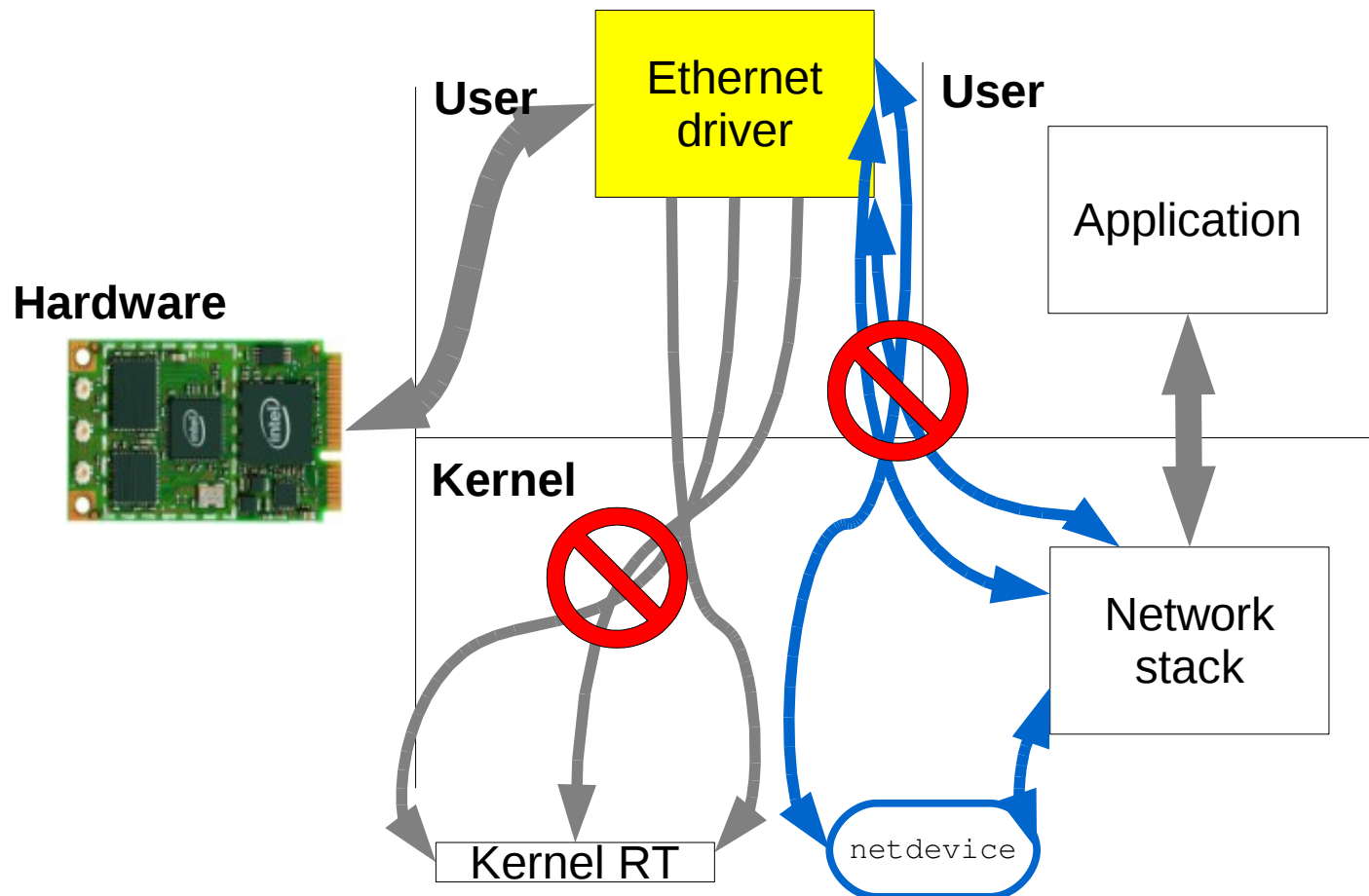


Current Linux driver architecture

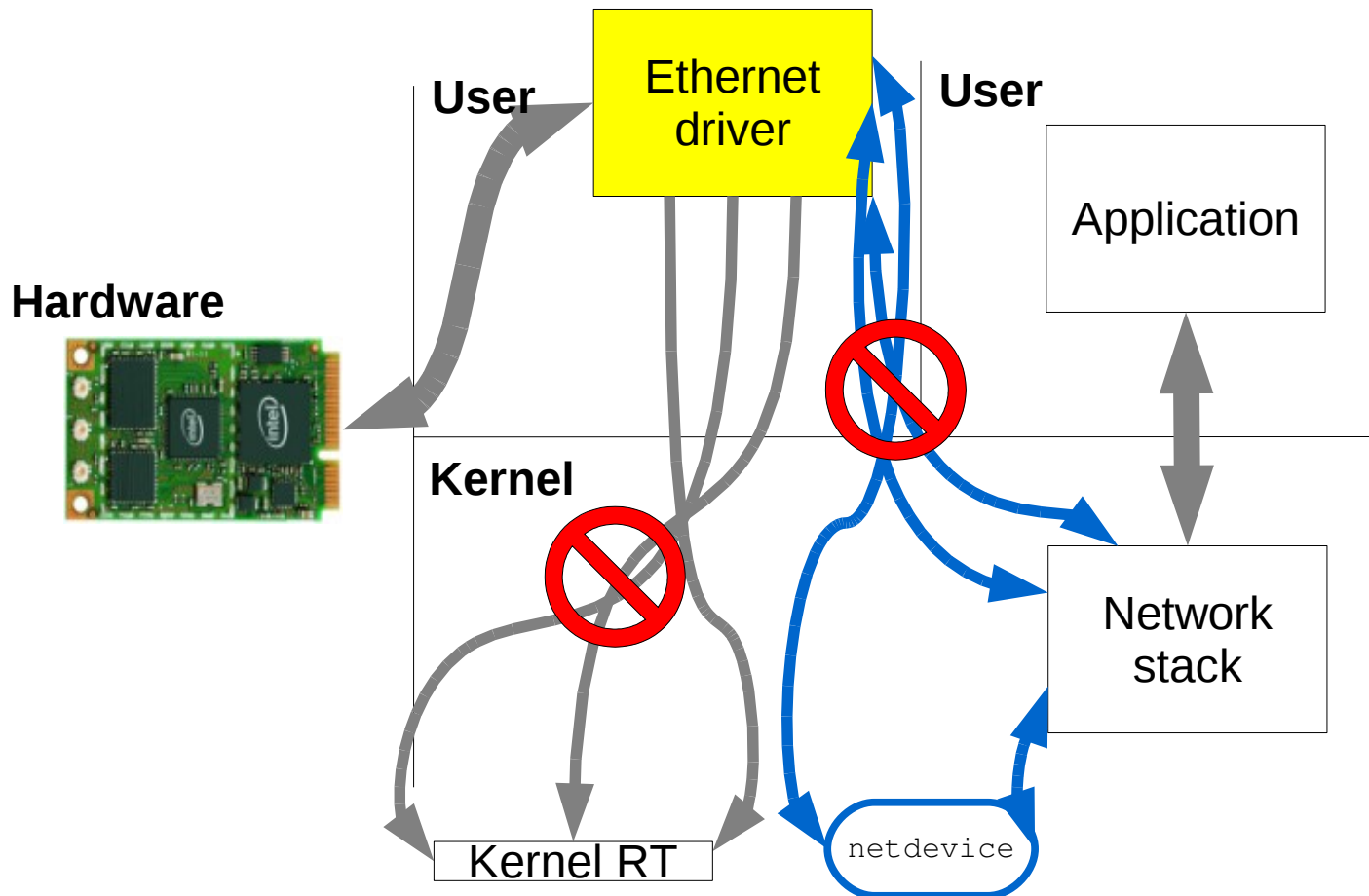


Linux user-space driver problem

Kernel RT and driver APIs won't work for untrusted drivers in a different AS

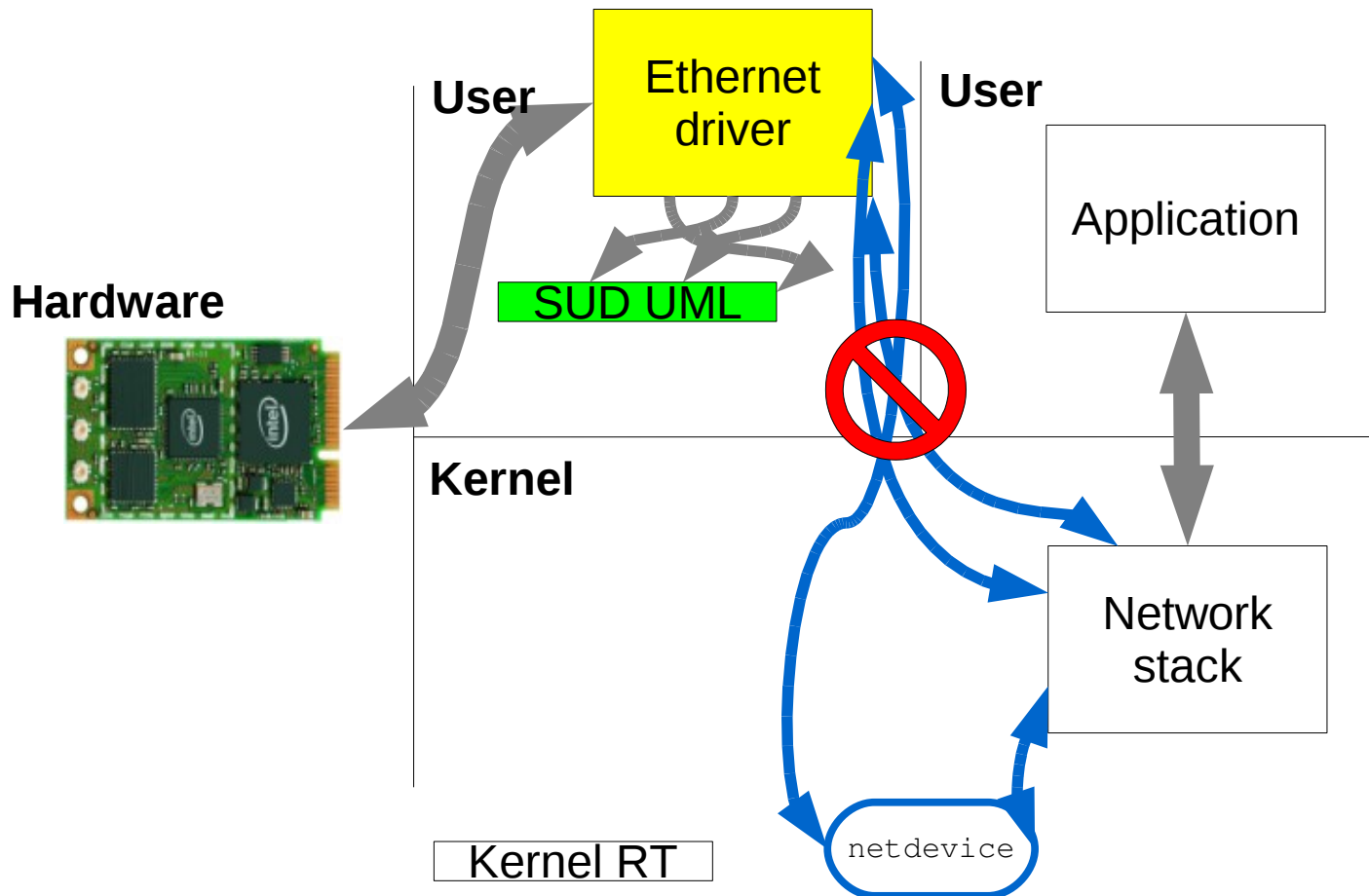


SUD's approach



SUD's approach

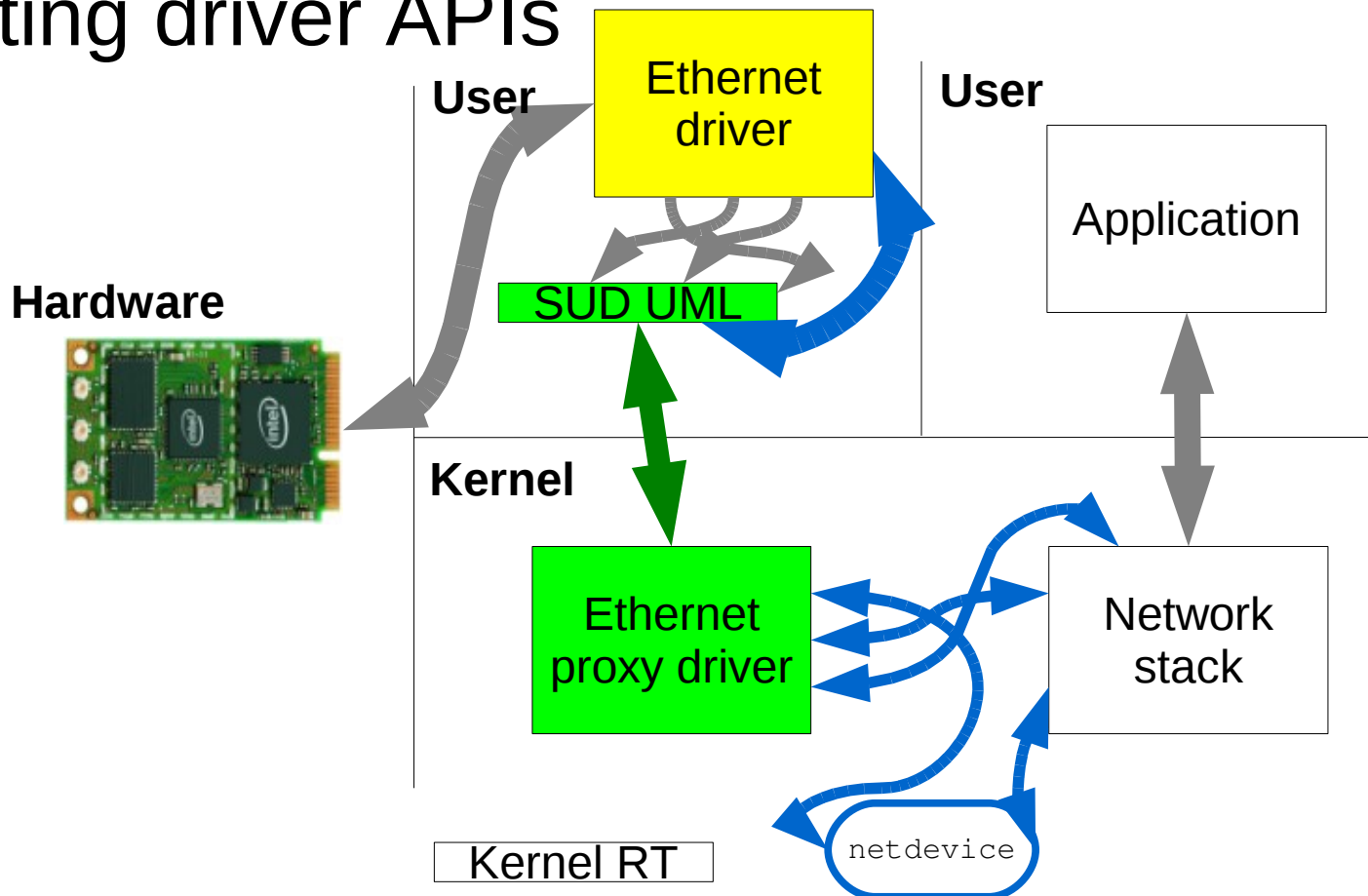
SUD UML handles calls to kernel RT



SUD's approach

SUD UML handles calls to kernel RT

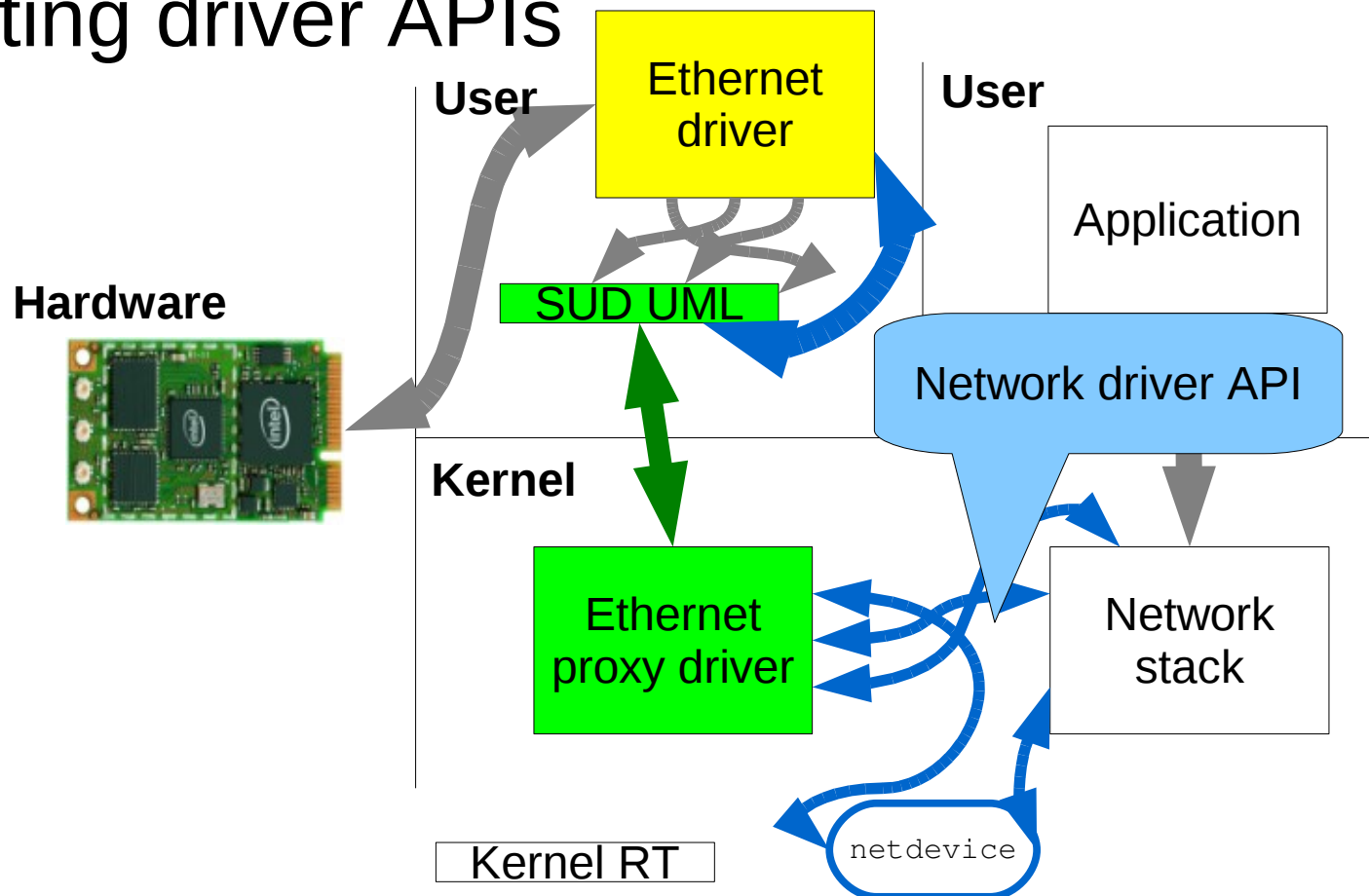
Proxy driver and SUD UML allow reuse of existing driver APIs



SUD's approach

SUD UML handles calls to kernel RT

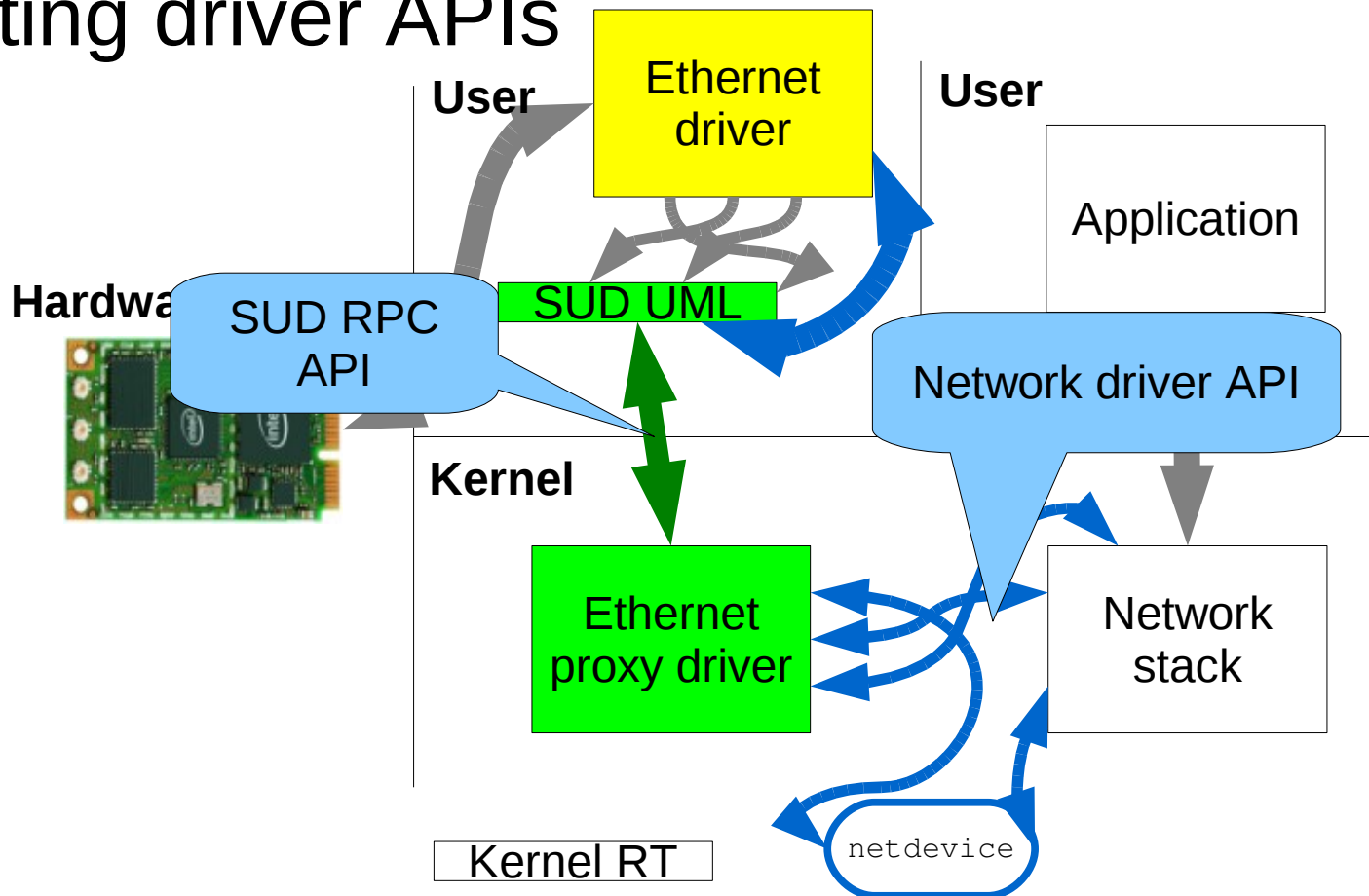
Proxy driver and SUD UML allow reuse of existing driver APIs



SUD's approach

SUD UML handles calls to kernel RT

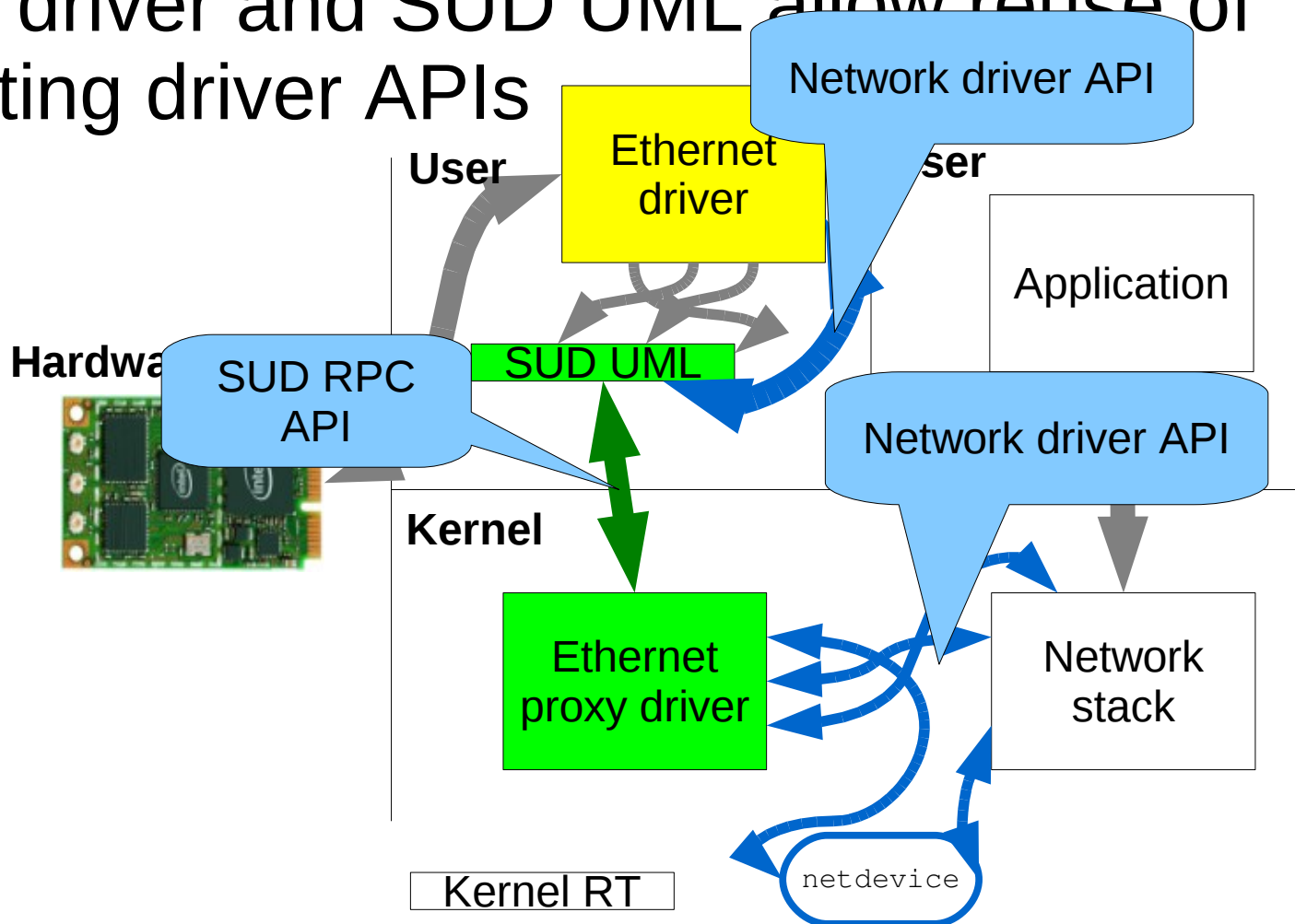
Proxy driver and SUD UML allow reuse of existing driver APIs



SUD's approach

SUD UML handles calls to kernel RT

Proxy driver and SUD UML allow reuse of existing driver APIs



SUD's results

Tolerate malicious device drivers

Proxy drivers small (~500 LOC)

One proxy driver per device class

Few kernel modifications (~50 LOC)

Unmodified drivers (6 test drivers)

High performance, low overhead

No need for new OS or language

Security challenge: prevent attacks

Problem: driver must perform privileged operations

Memory access, driver API, DMA, interrupts, ...

Attacks from driver code:

Direct system attacks: memory corruption, ...

Driver API attacks: invalid return value, deadlock, ...

Attacks from device:

DMA to DRAM, peer-to-peer attacks, interrupt storms

Practical challenges

High performance, low overhead

Challenge: interact with hardware and kernel at high rate, kernel-user switch expensive

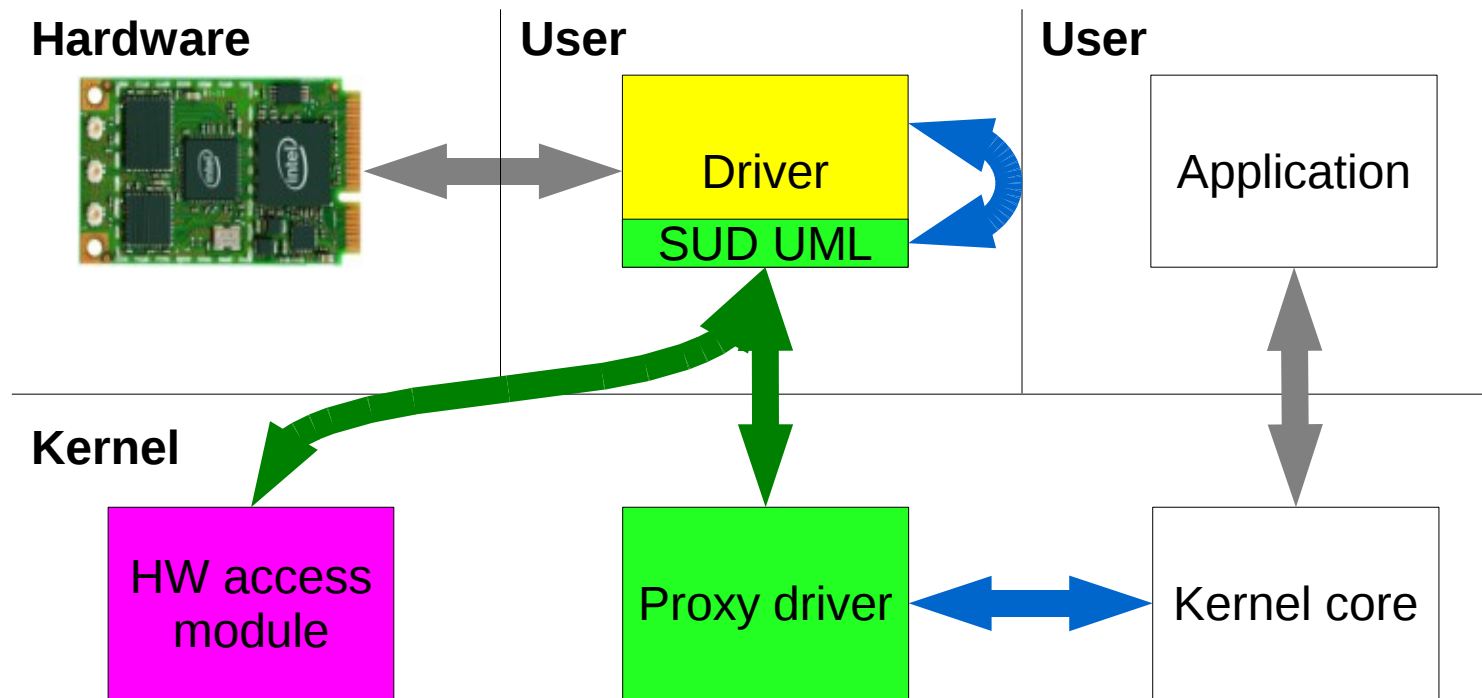
E.g. Ethernet driver ~100k times a second

Reuse existing drivers and kernel

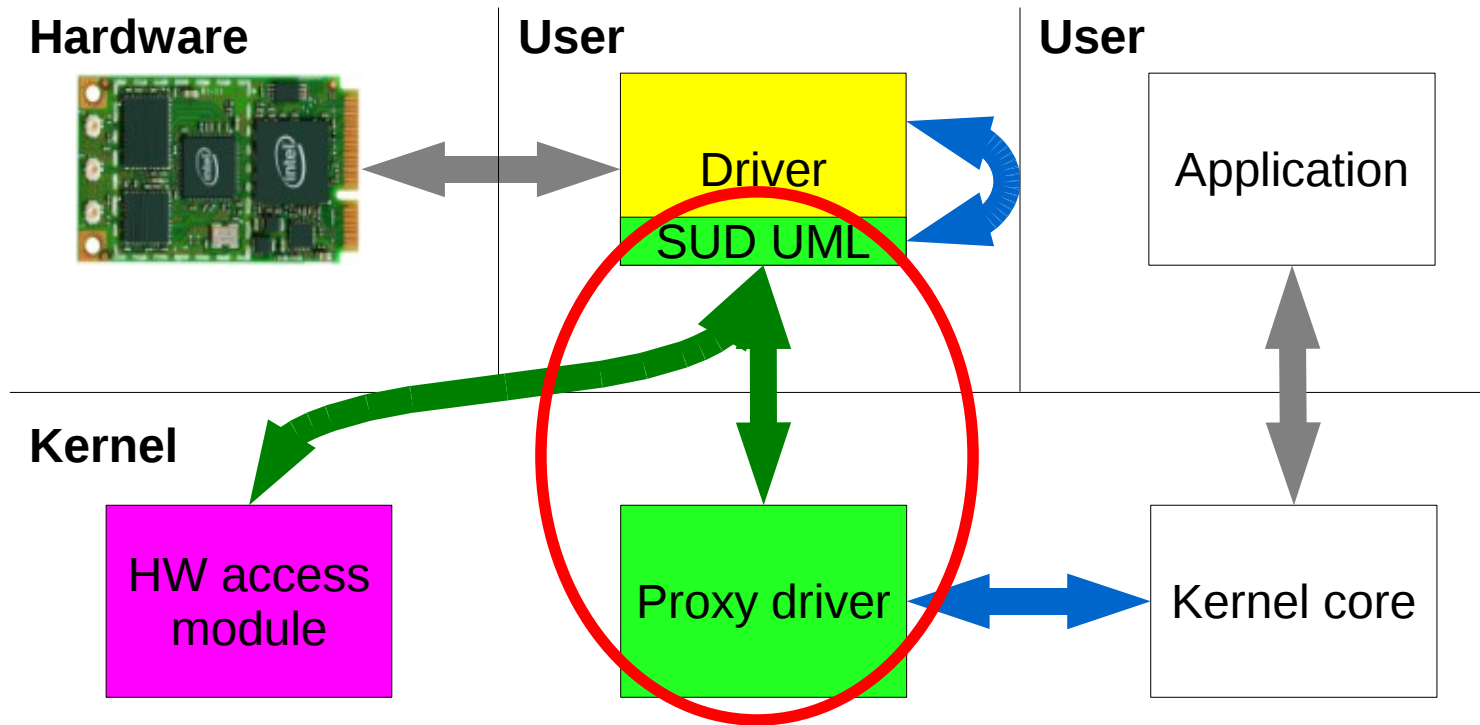
Challenge: drivers assume fully-privileged kernel env.

Challenge: kernel driver API complex, non-uniform

SUD overview



SUD overview



Linux driver APIs

Linux defines a driver API for each device class

Driver and kernel functions and variables

Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```
struct wireless_ops {
    int (*tx)(struct sk_buff*);
    int (*configure_filter)(int);
    ...
};

struct wireless_hw {
    int conf;
    int flags
    ....
};
```

Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```
struct wireless_ops {  
    int (*tx)(struct sk_buff*);  
    int (*configure_filter)(int);  
    ...  
};  
  
struct wireless_hw {  
    int conf;  
    int flags  
    ....  
};
```

Proxy drivers and SUD-UML convert API to RPCs

Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```
struct wireless_ops {  
    int (*tx)(struct sk_buff*);  
    int (*configure_filter)(int);  
    ...  
};  
  
struct wireless_hw {  
    int conf;  
    int flags  
    ....  
};
```

Proxy drivers and SUD-UML convert API to RPCs

Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```
struct wireless_ops {  
    int (*tx)(struct sk_buff*);  
    int (*configure_filter)(int);  
    ...  
};  
  
struct wireless_hw {  
    int conf;  
    int flags  
    ....  
};
```

Called in a non-preemptible context

Proxy drivers and SUD-UML convert API to RPCs

Example: wireless driver API

Linux defines a driver API for each device class

Driver and kernel functions and variables

```
struct wireless_ops {  
    int (*tx)(struct sk_buff*);  
    int (*configure_filter)(int);  
    ...  
};  
struct wireless_hw {  
    int conf;  
    int flags  
    ....  
};
```

Called in a non-preemptible context

Driver API variable

Proxy drivers and SUD-UML convert API to RPCs

Wireless driver in SUD

Basic driver API → SUD RPC API → driver API

Non-preemptable function: implement in proxy

Driver API variable: shadow variables

Example 1: transmit a packet

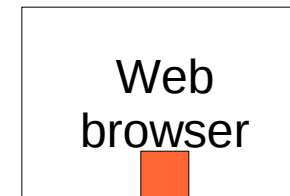
Hardware



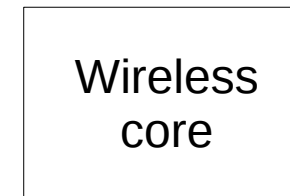
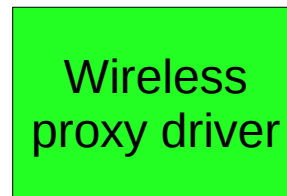
User



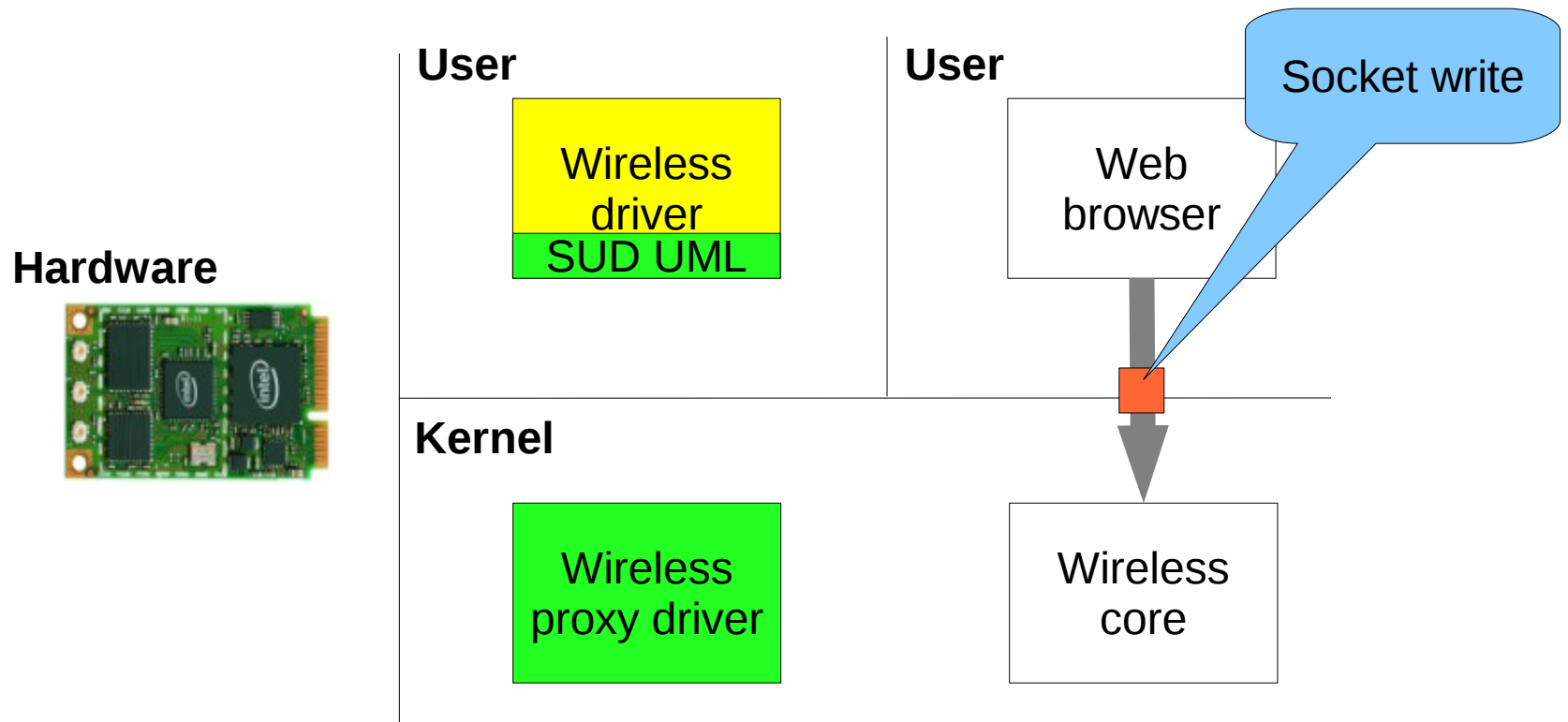
User



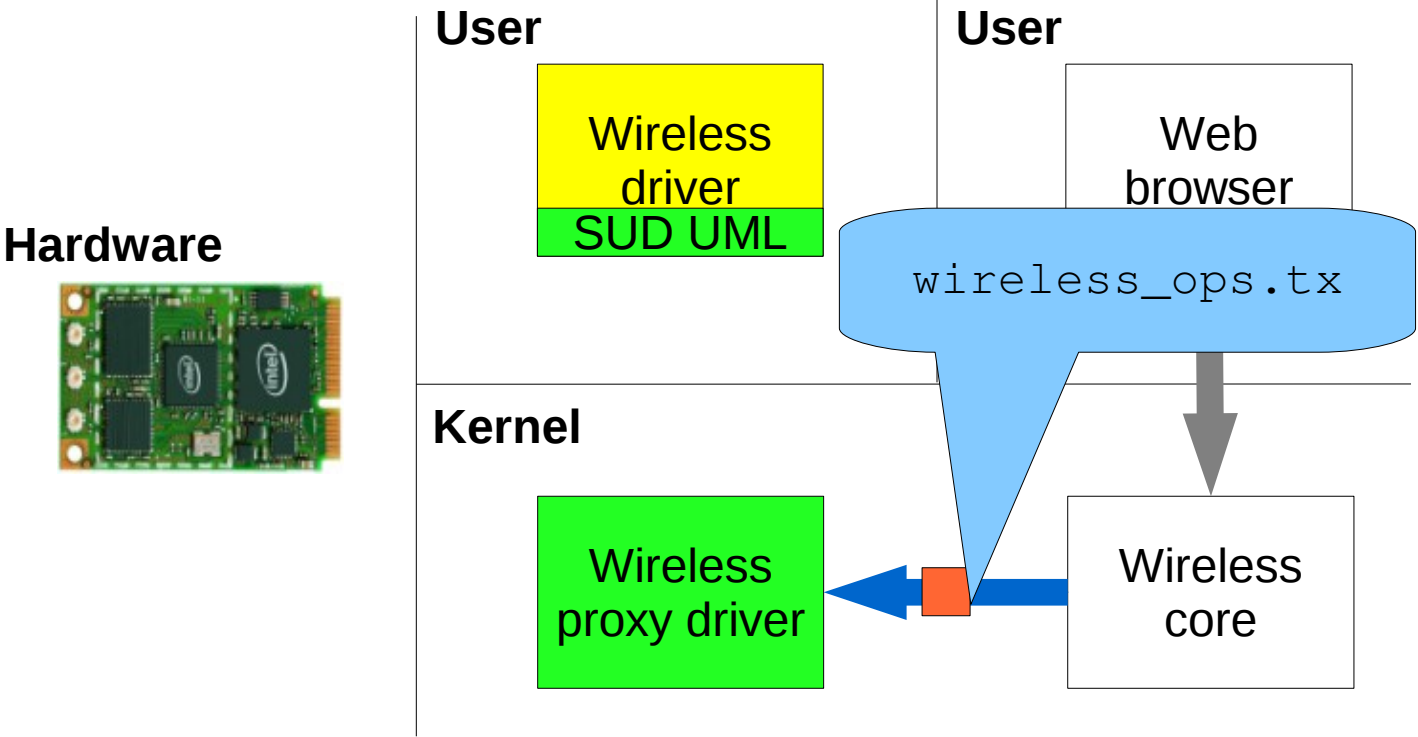
Kernel



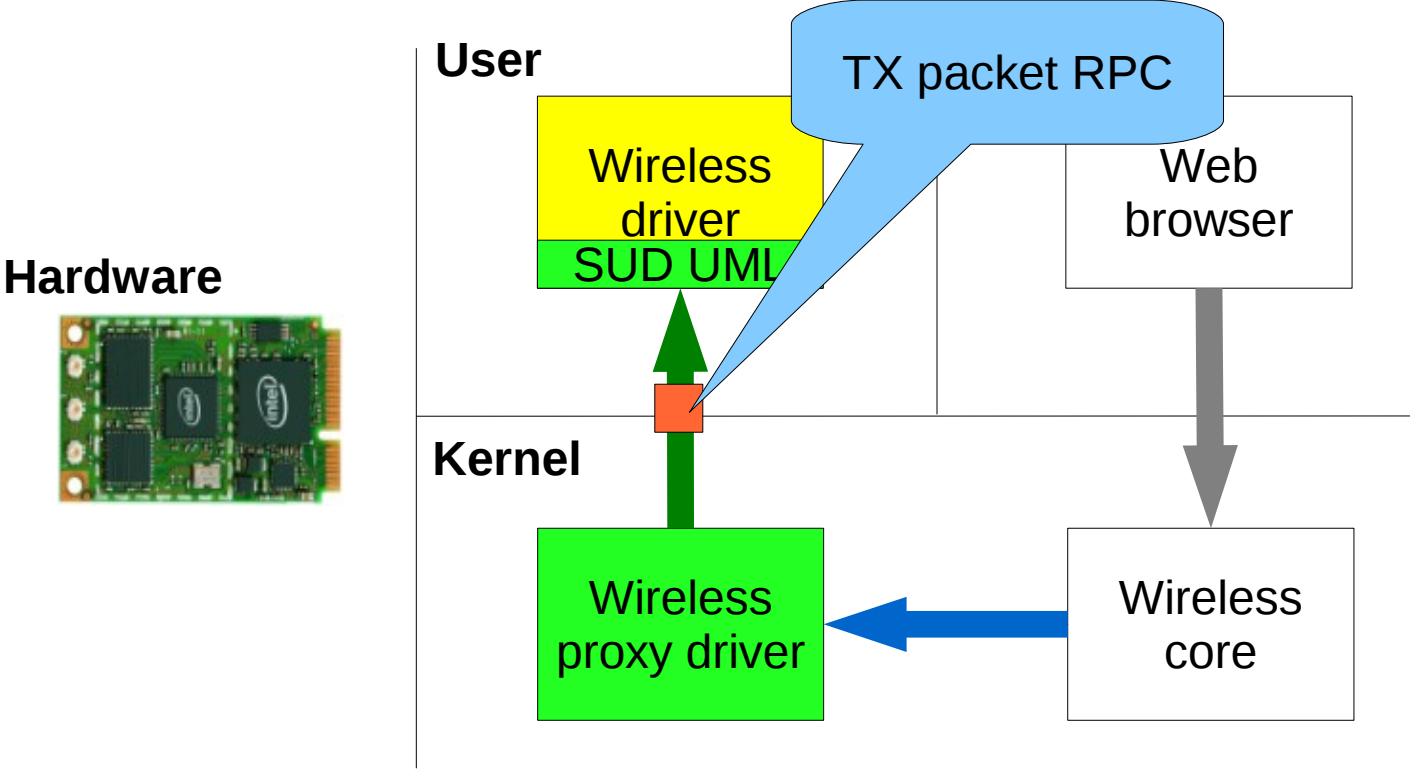
Example 1: transmit a packet



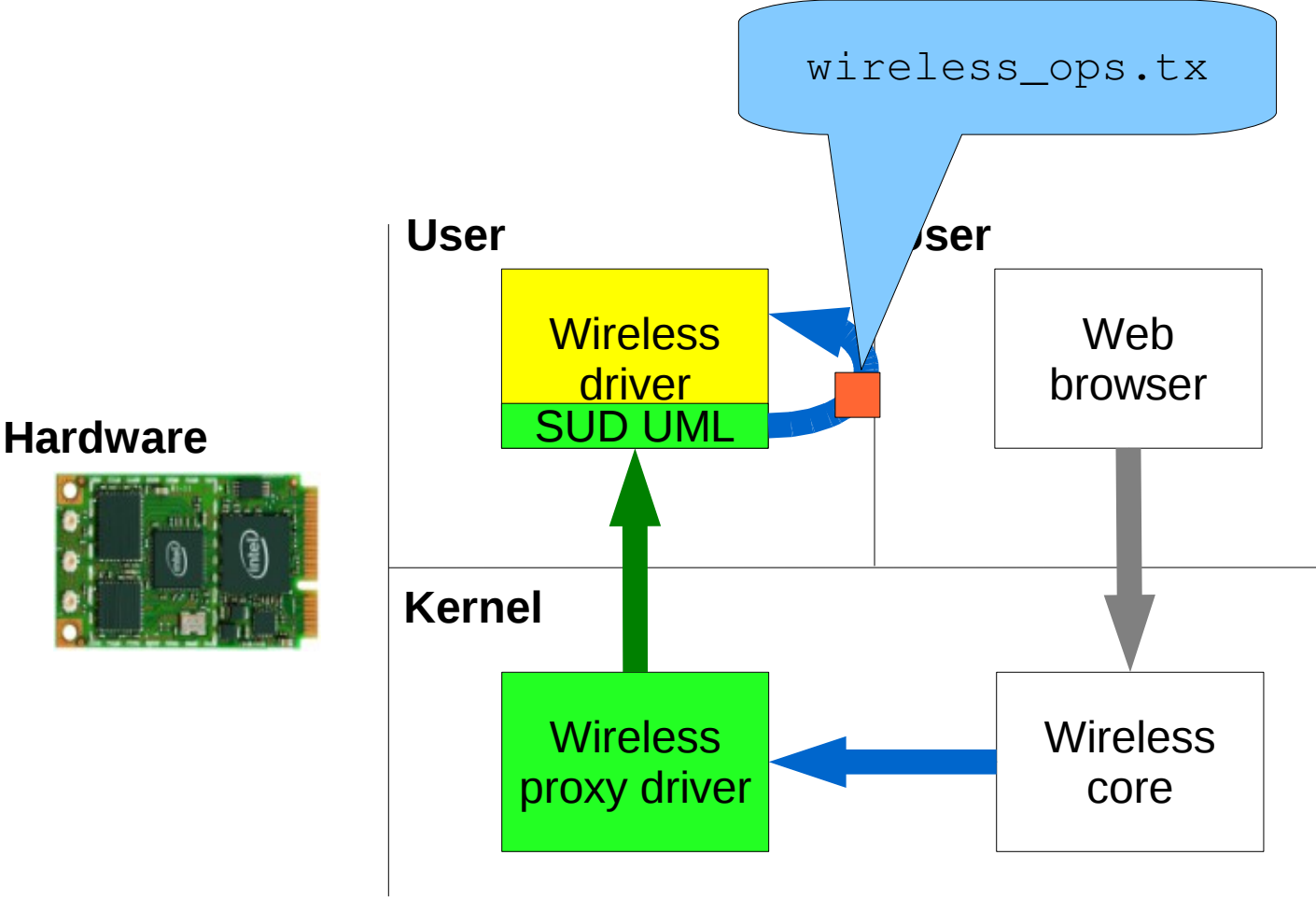
Example 1: transmit a packet



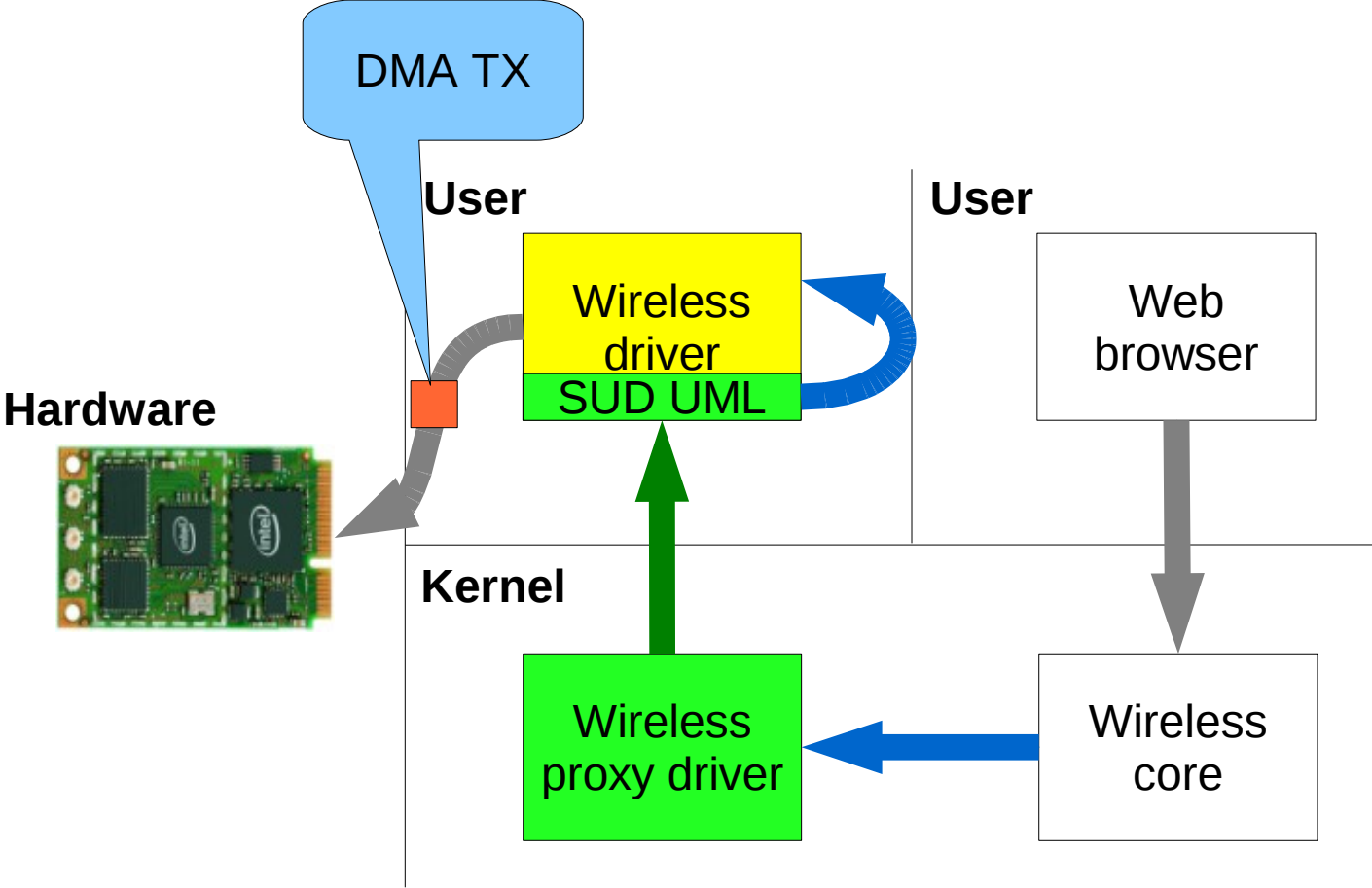
Example 1: transmit a packet



Example 1: transmit a packet

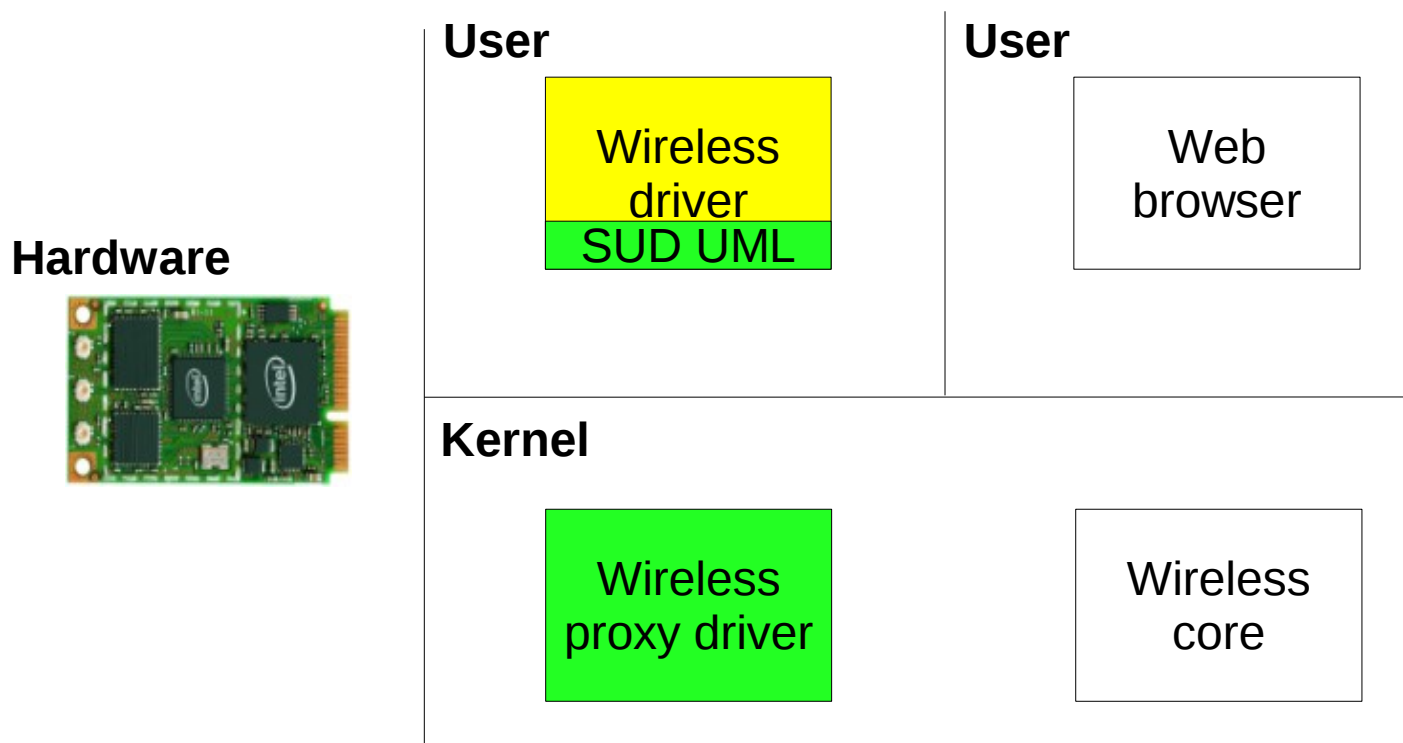


Example 1: transmit a packet



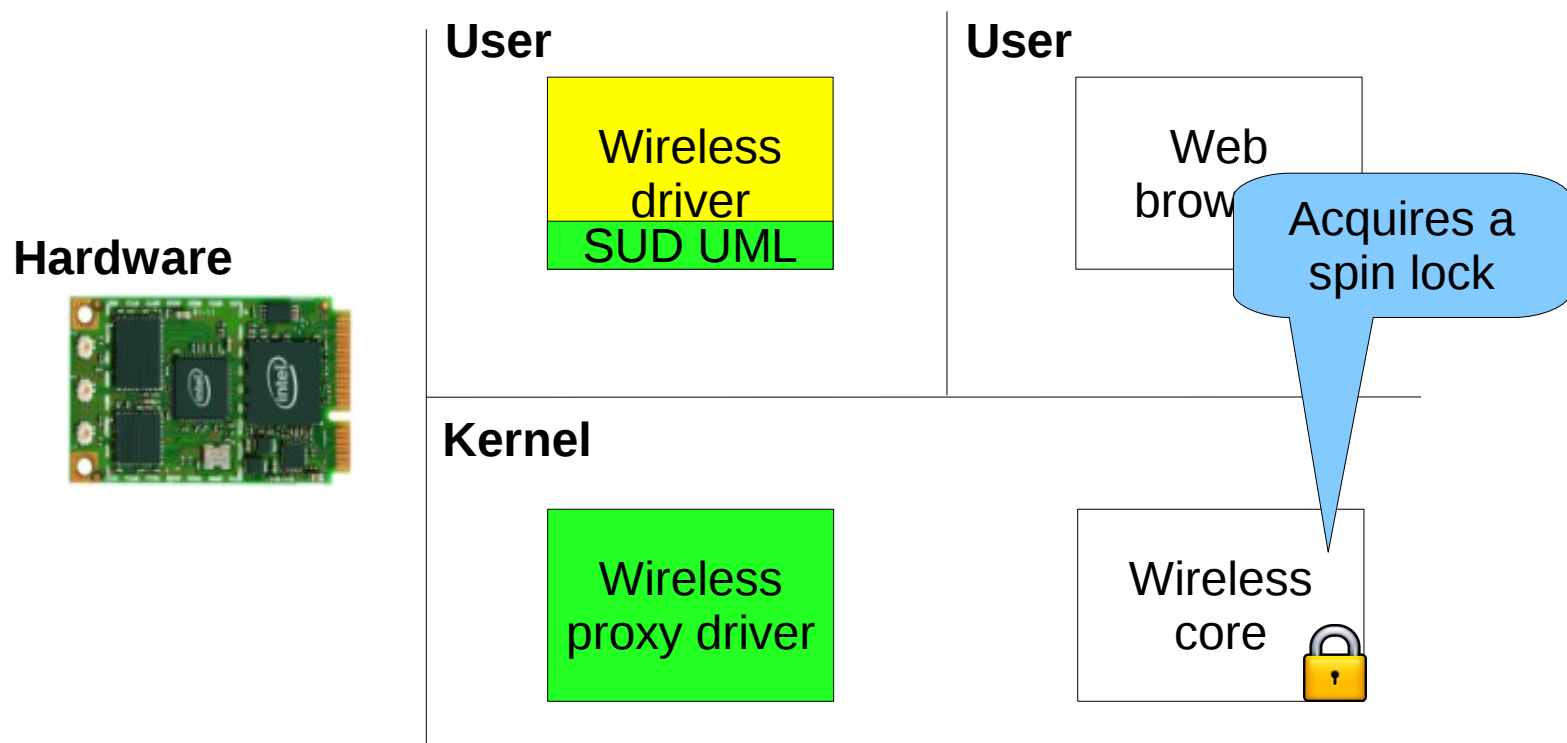
Example 2: non-preemptable callback

Problem: unable to switch to user-space



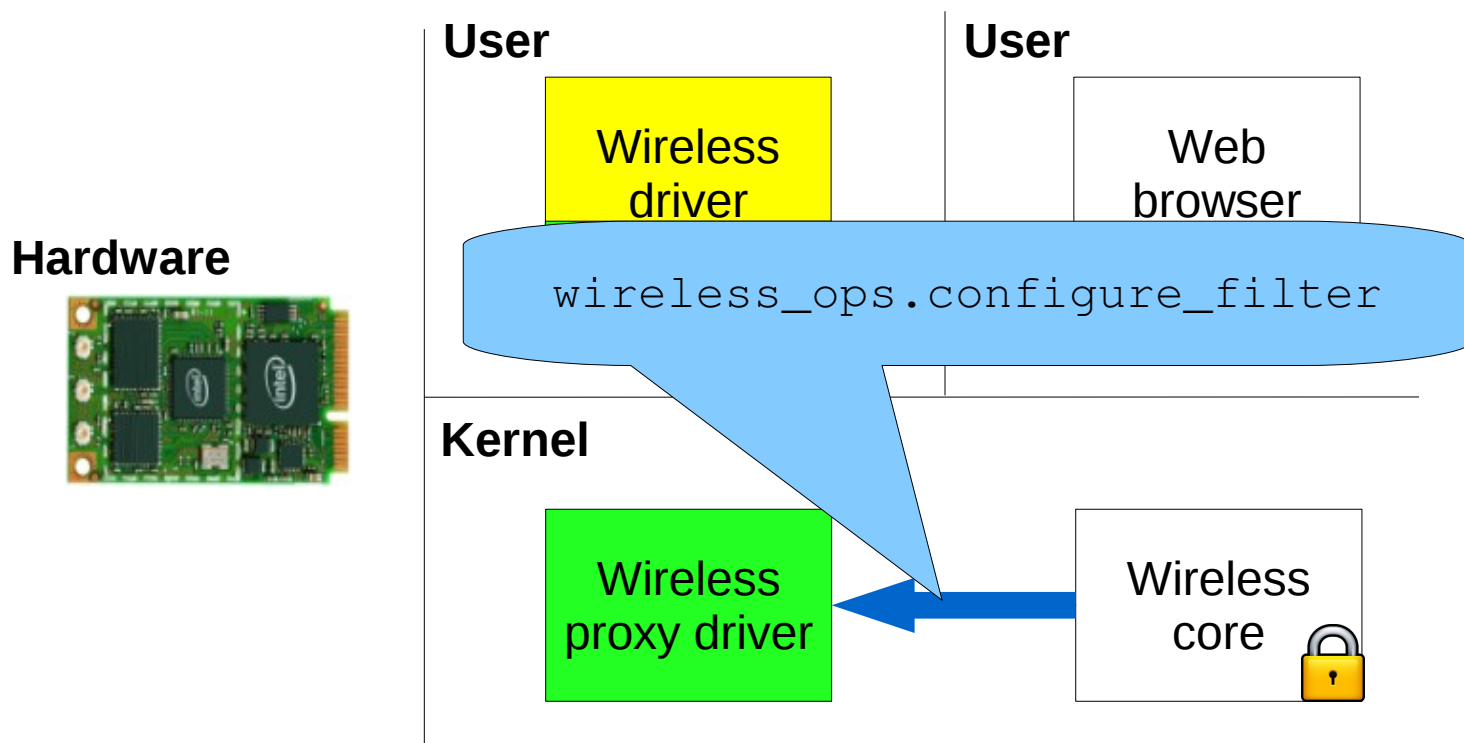
Example 2: non-preemptable callback

Problem: unable to switch to user-space



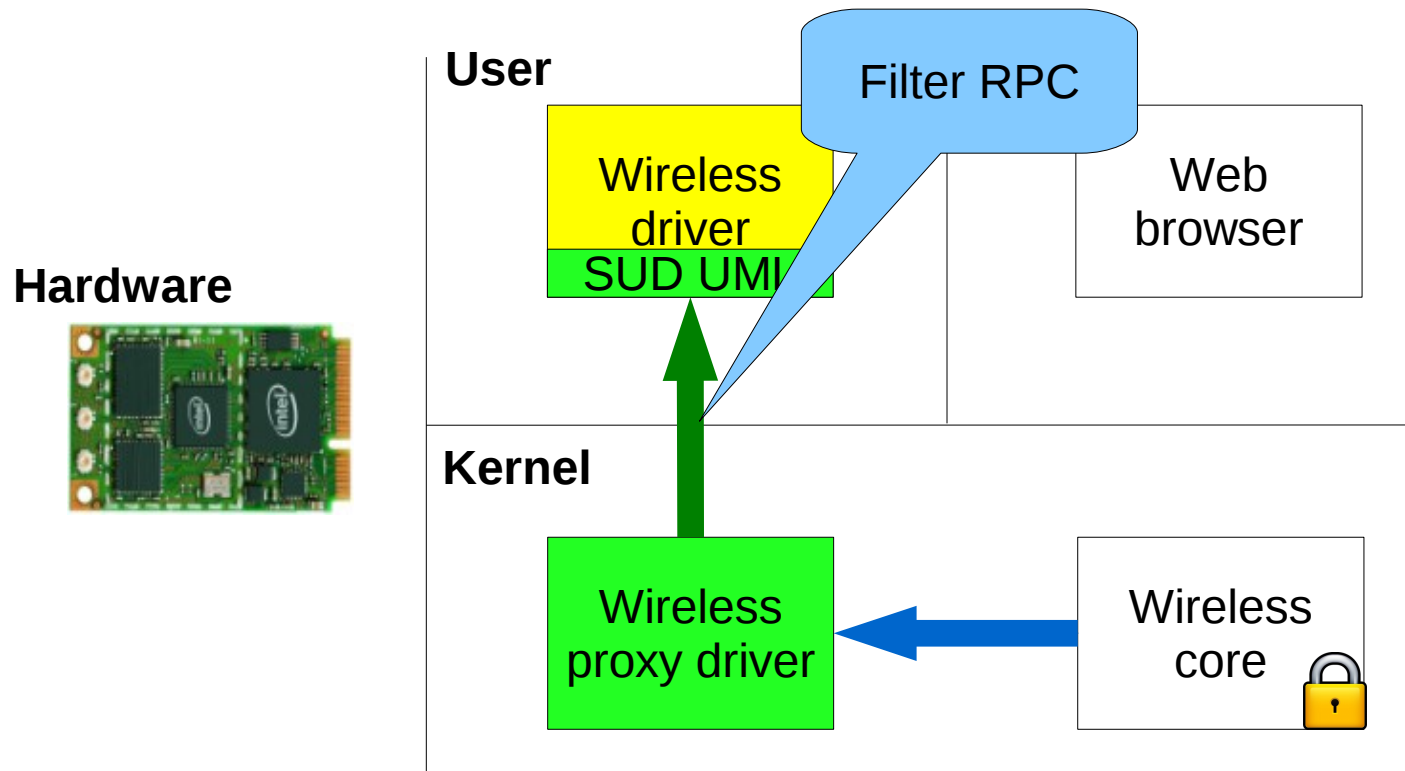
Example 2: non-preemptable callback

Problem: unable to switch to user-space



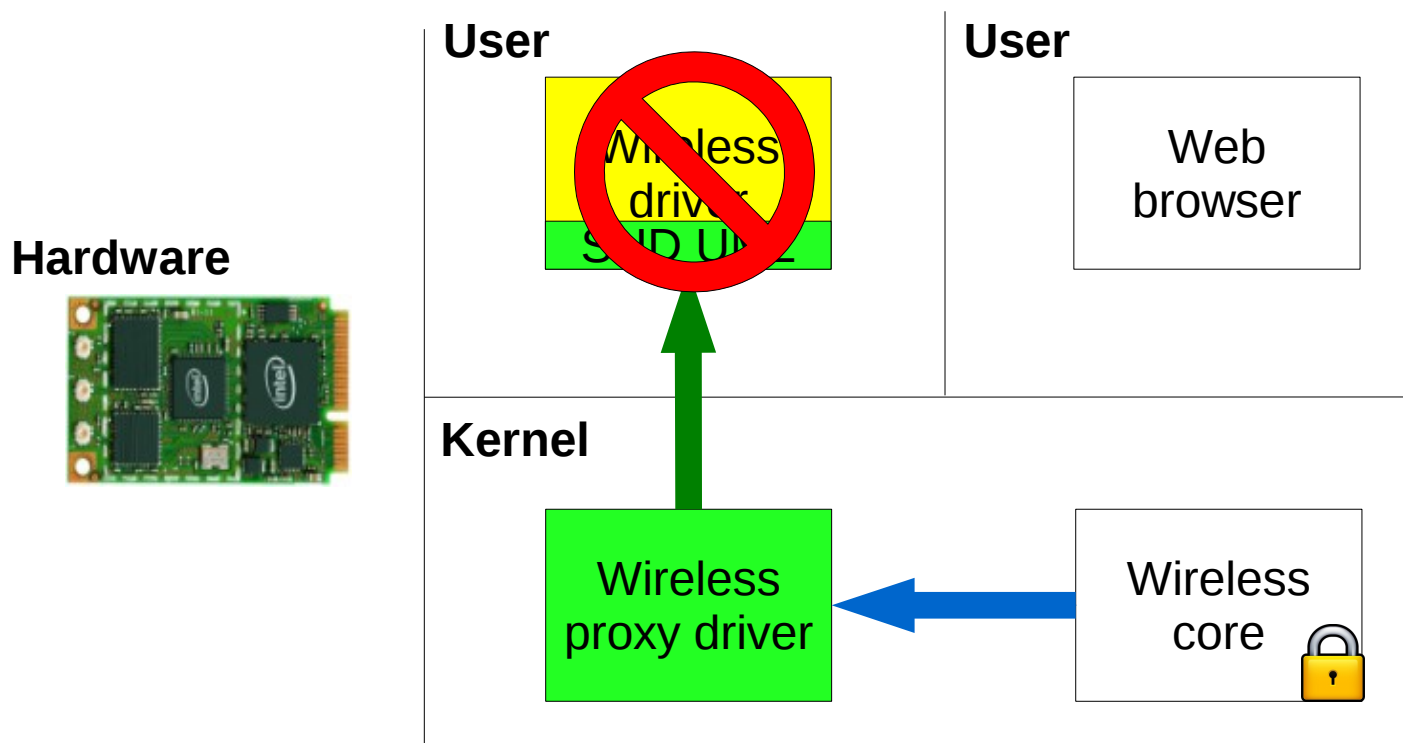
Example 2: non-preemptable callback

Problem: unable to switch to user-space



Example 2: non-preemptable callback

Problem: unable to switch to user-space

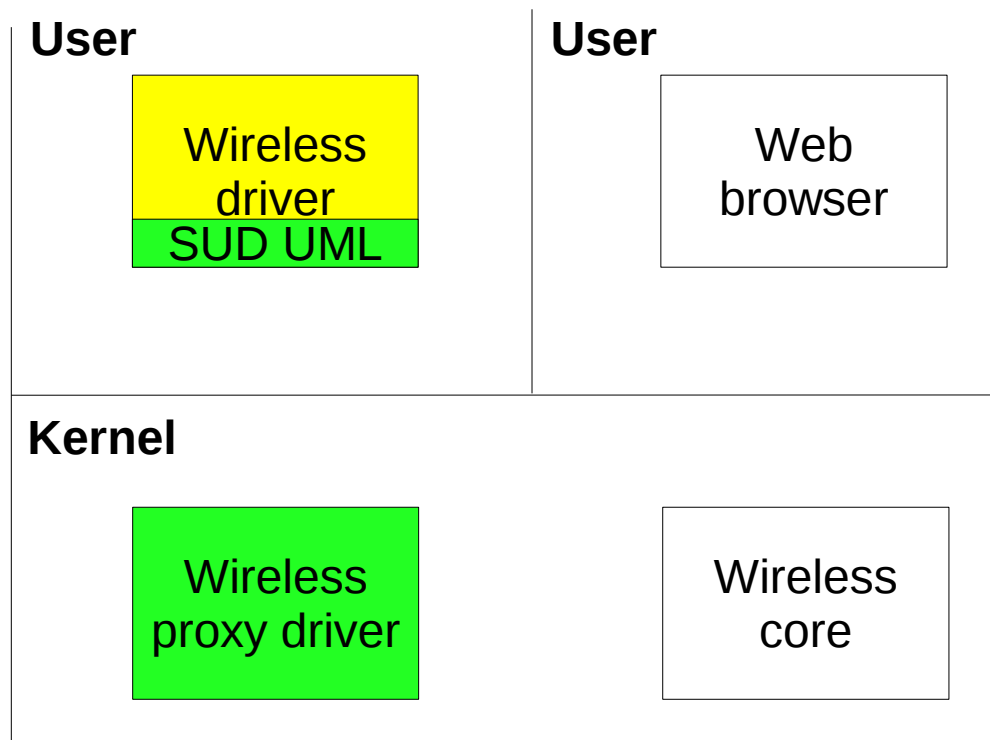


Example 2: non-preemptable callback

Problem: unable to switch to user-space

Solution: implement directly in proxy driver

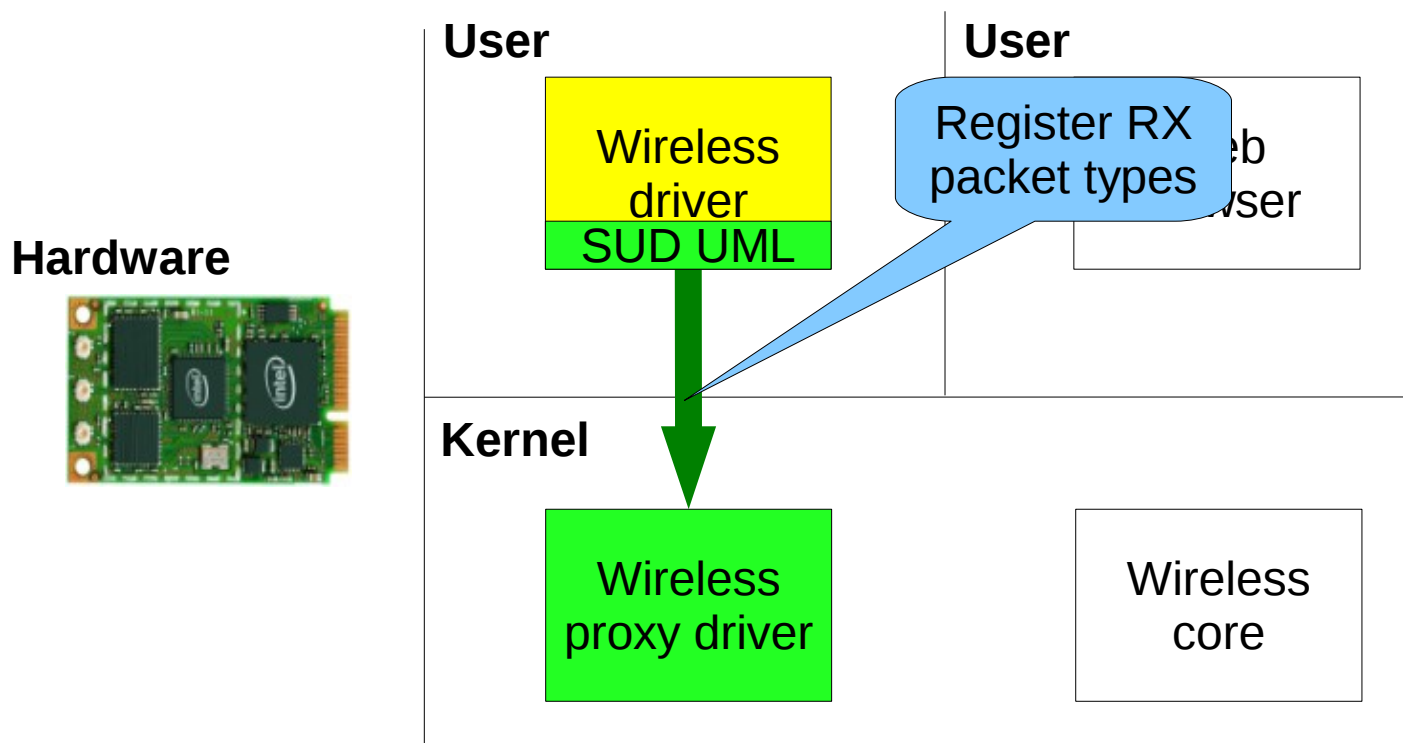
Hardware



Example 2: non-preemptable callback

Problem: unable to switch to user-space

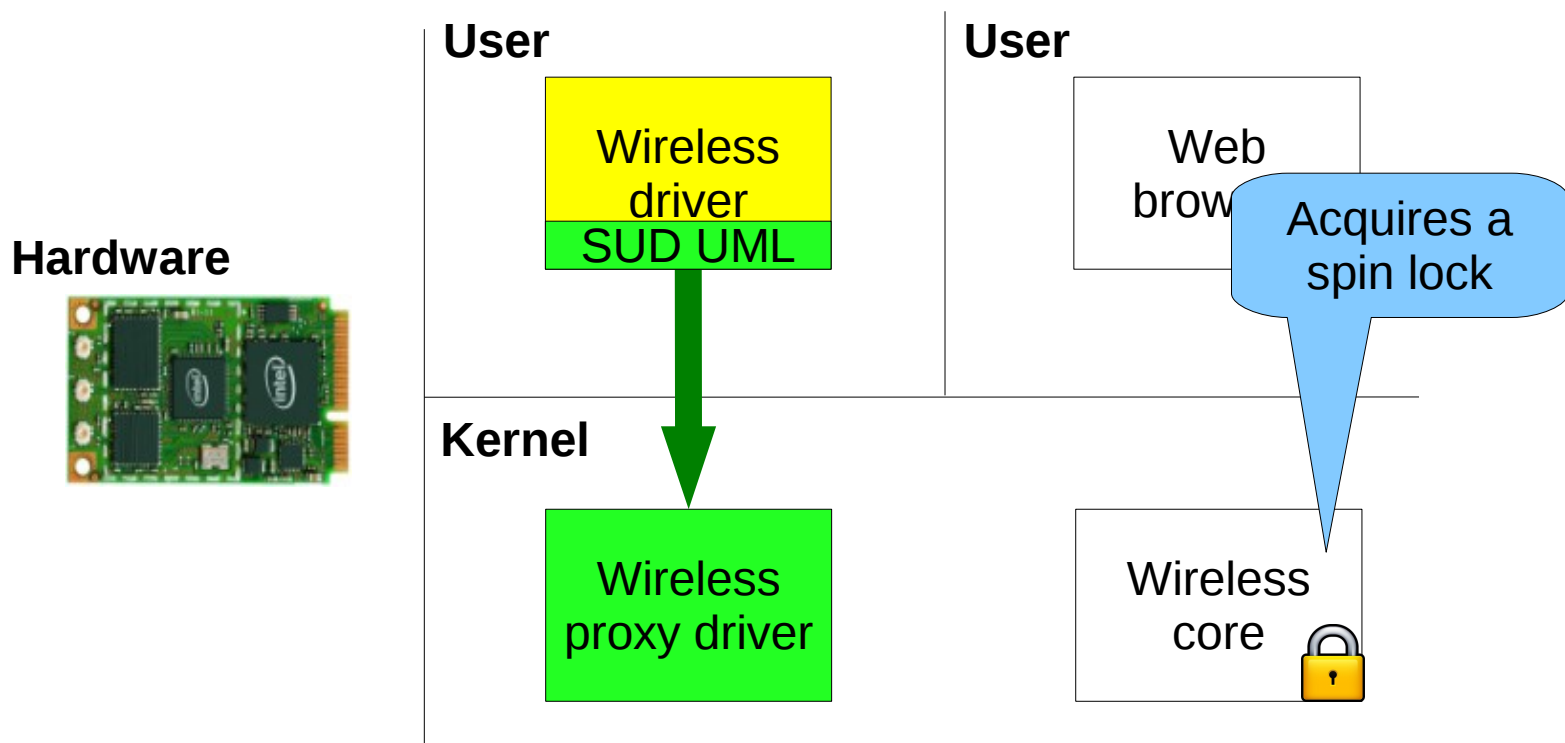
Solution: implement directly in proxy driver



Example 2: non-preemptable callback

Problem: unable to switch to user-space

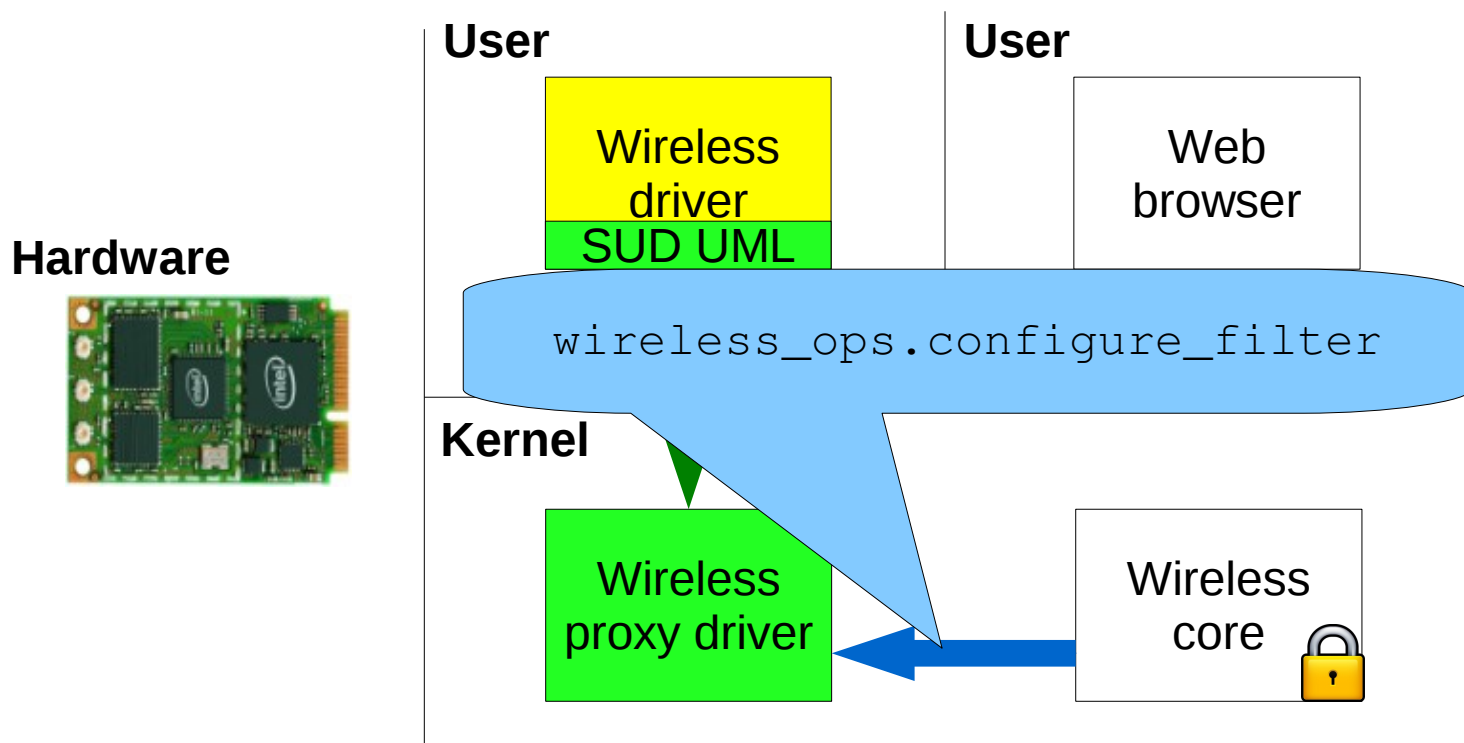
Solution: implement directly in proxy driver



Example 2: non-preemptable callback

Problem: unable to switch to user-space

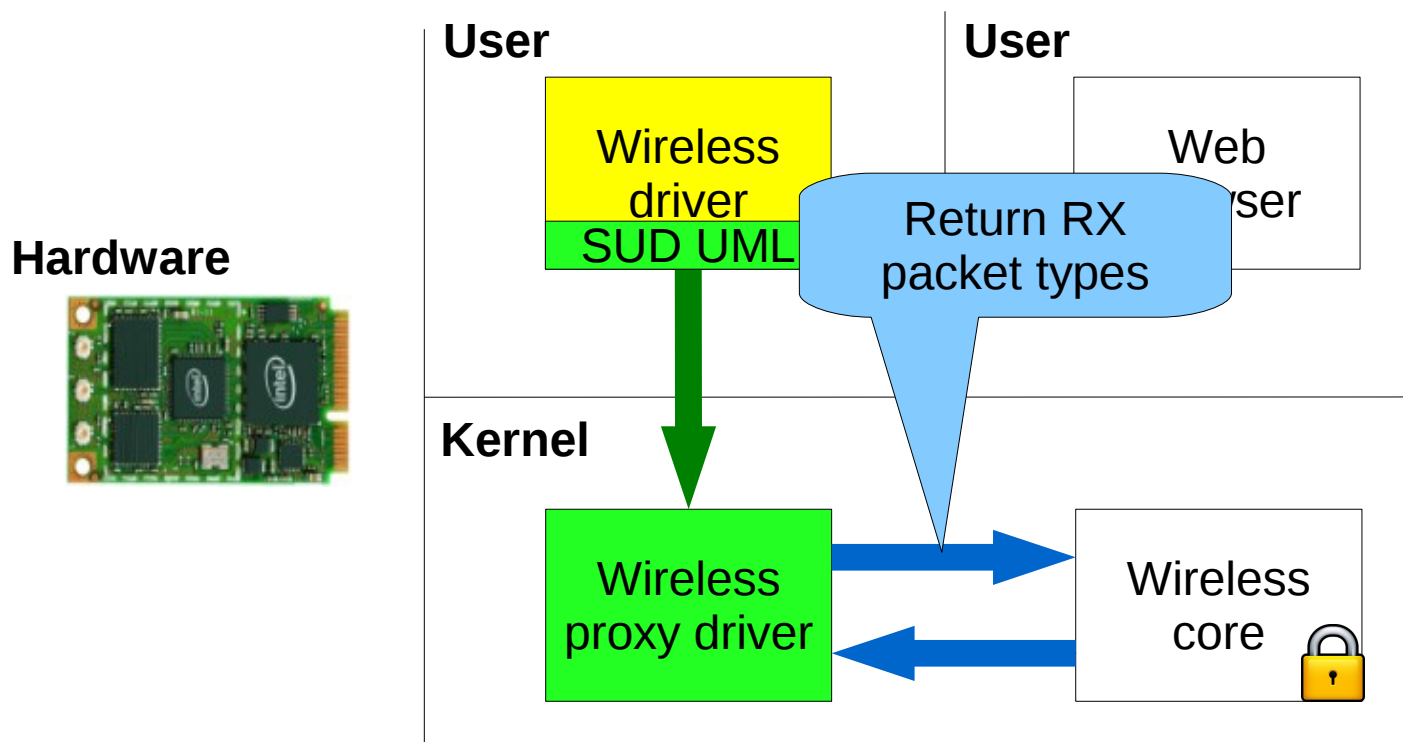
Solution: implement directly in proxy driver



Example 2: non-preemptable callback

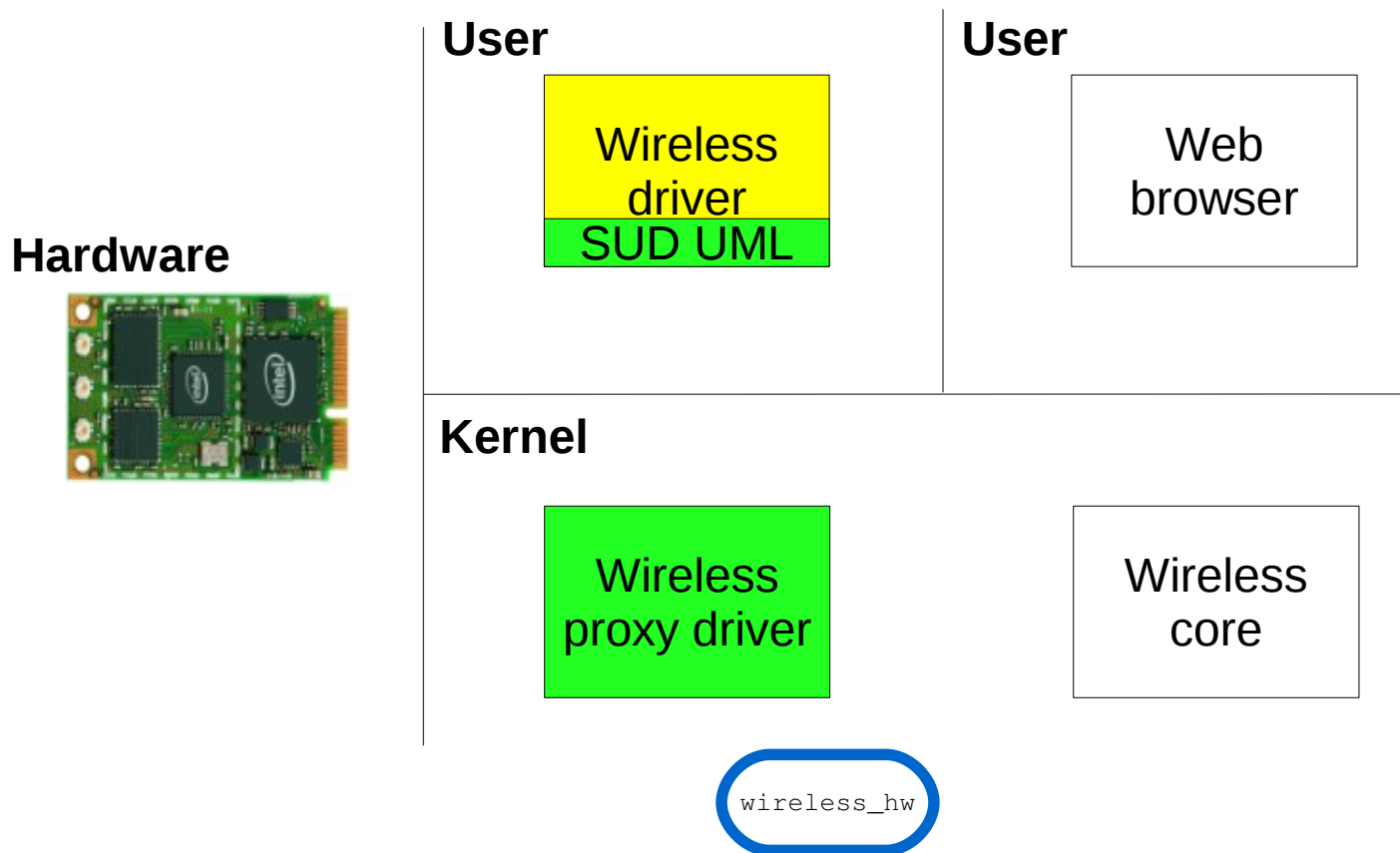
Problem: unable to switch to user-space

Solution: implement directly in proxy driver



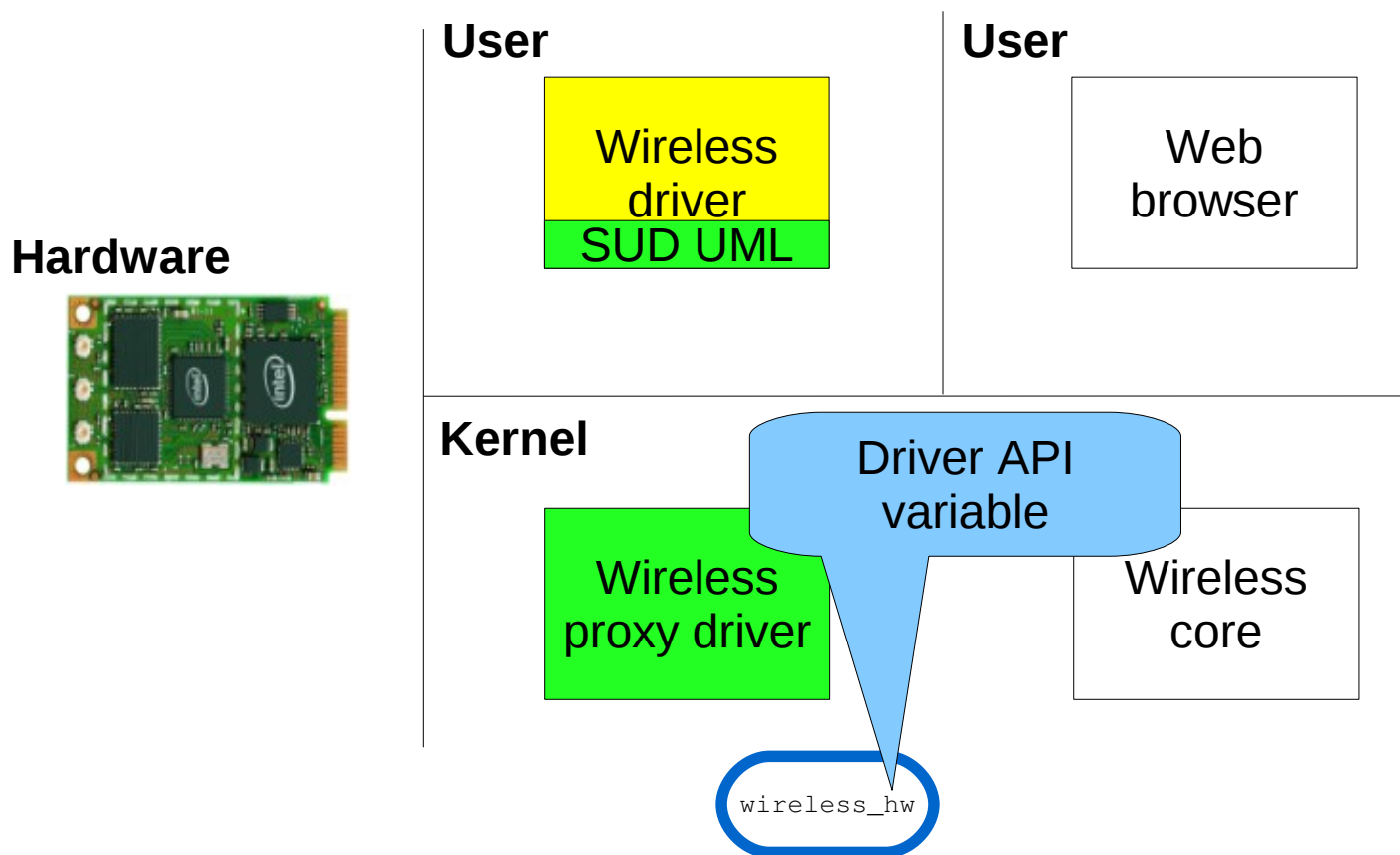
Example 3: driver API variables

Problem: user-space can't access API variables



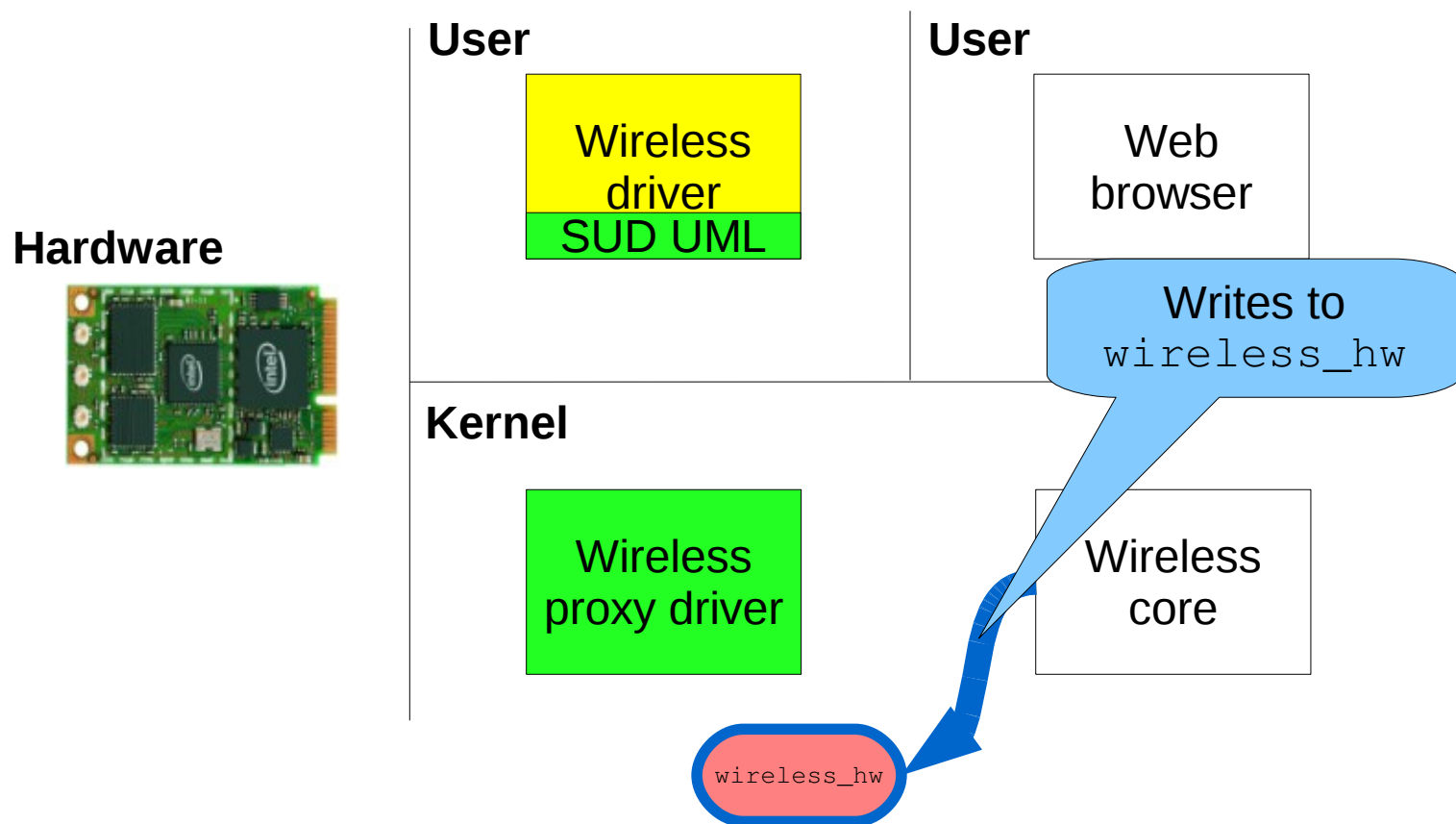
Example 3: driver API variables

Problem: user-space can't access API variables



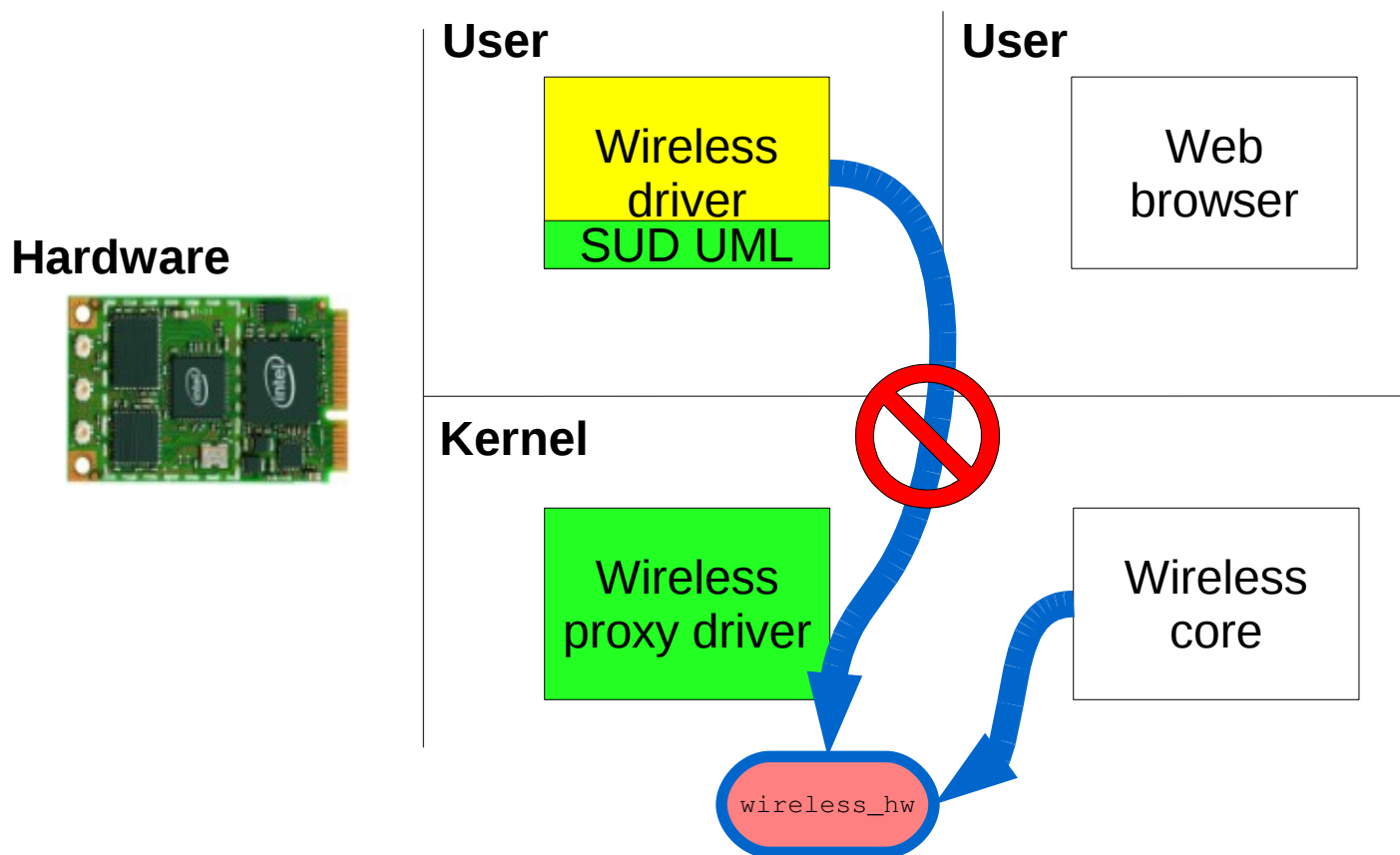
Example 3: driver API variables

Problem: user-space can't access API variables



Example 3: driver API variables

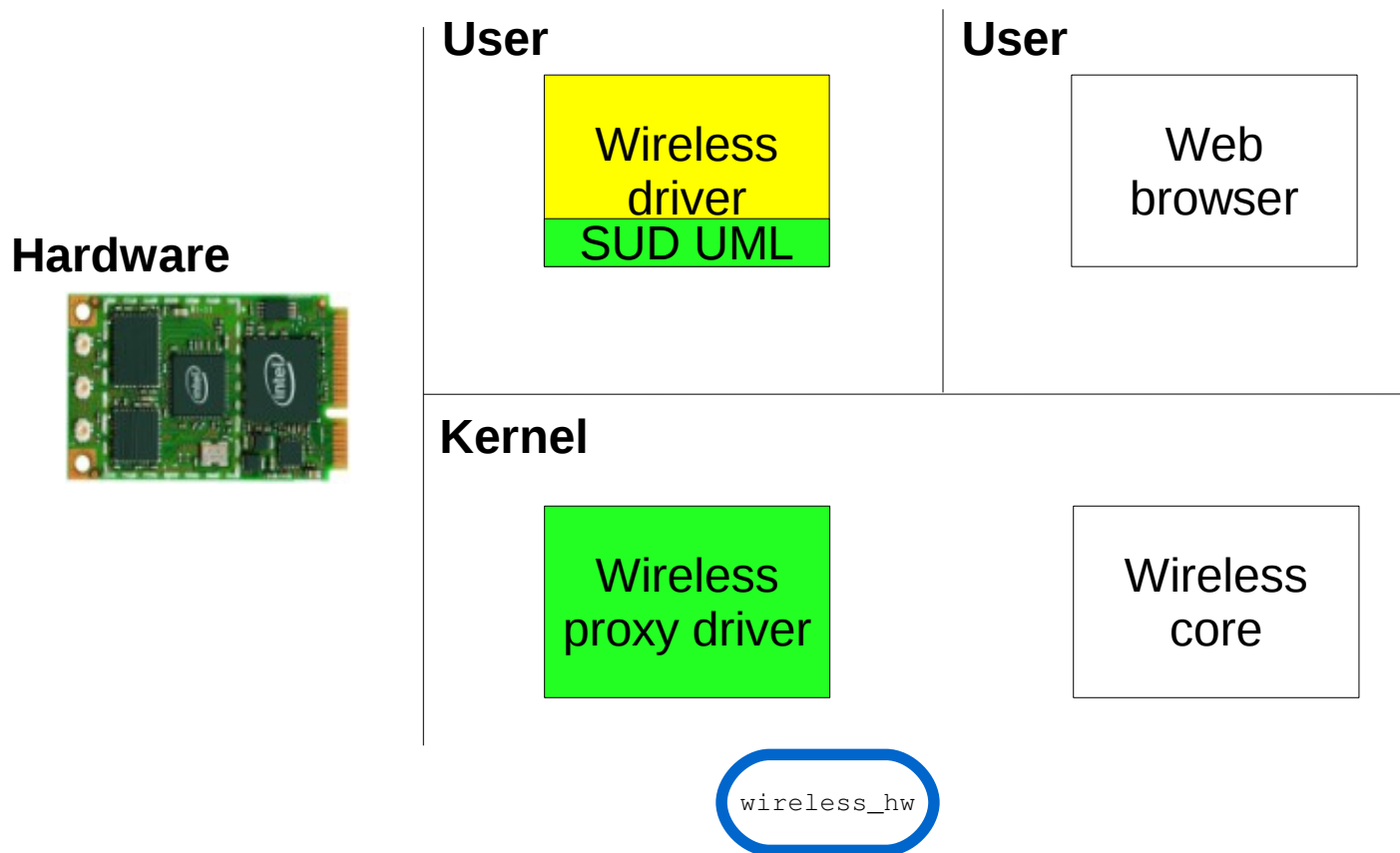
Problem: user-space can't access API variables



Example 3: driver API variables

Problem: user-space can't access API variables

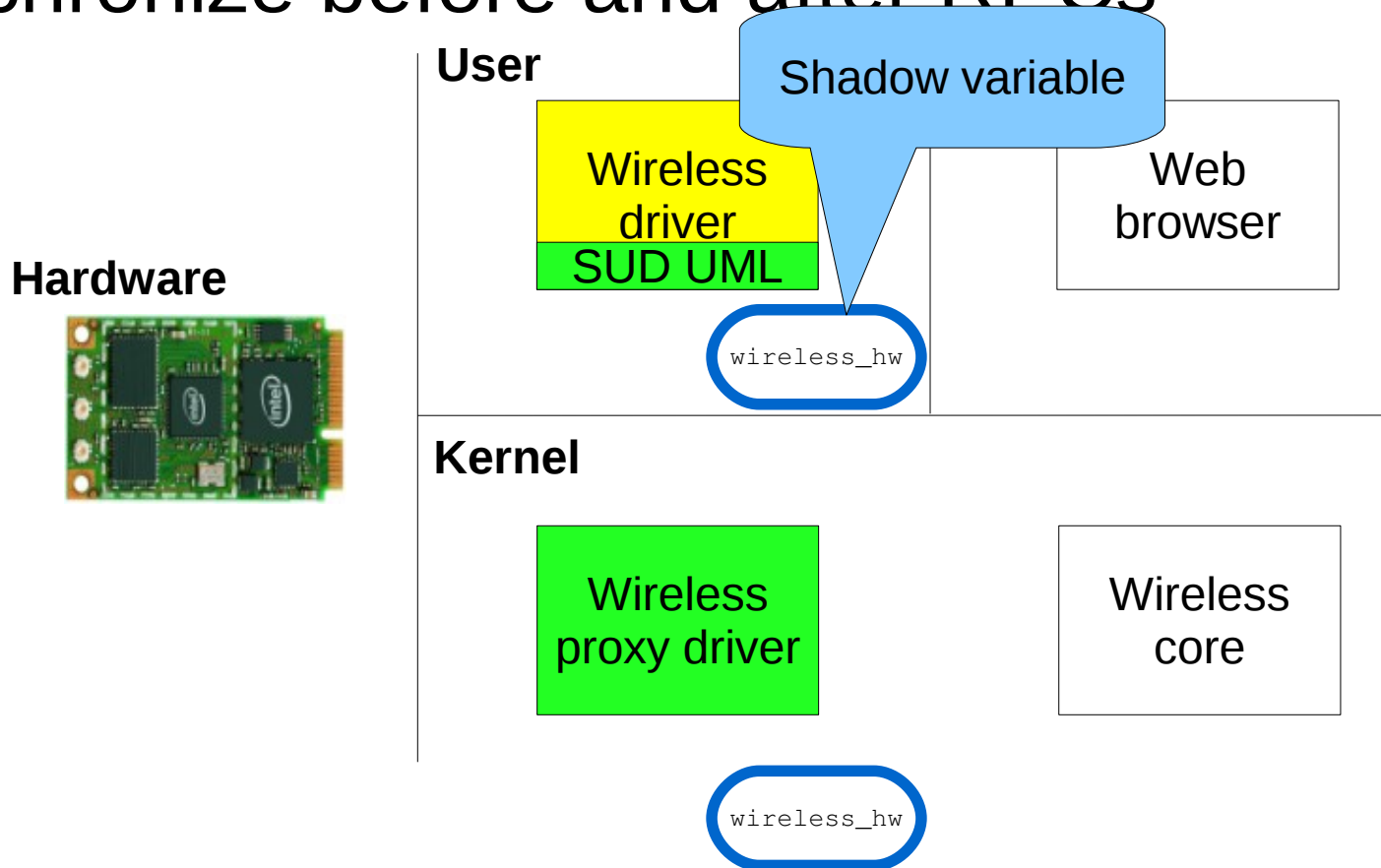
Solution: allocate a shadow copy and synchronize before and after RPCs



Example 3: driver API variables

Problem: user-space can't access API variables

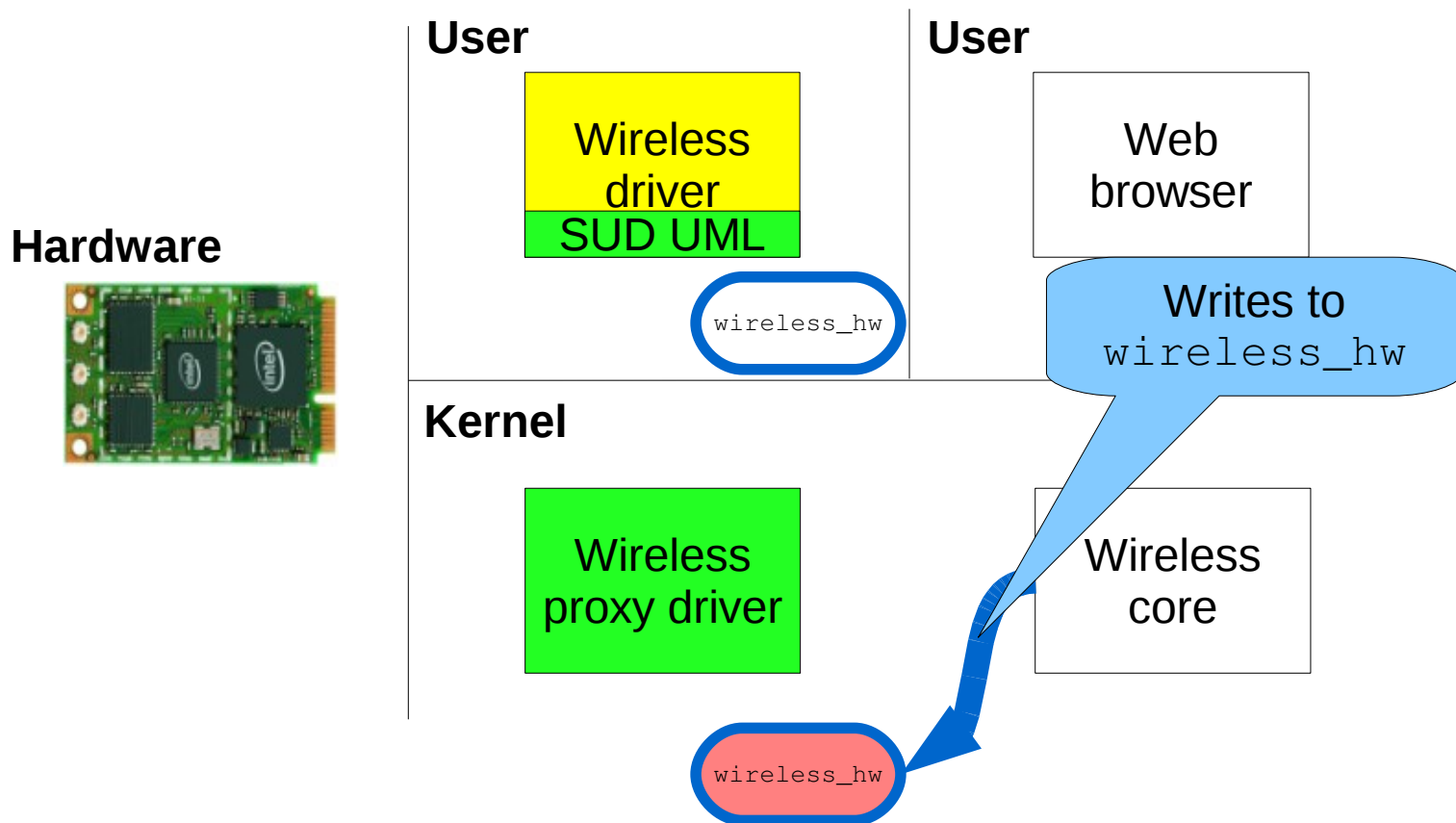
Solution: allocate a shadow copy and synchronize before and after RPCs



Example 3: driver API variables

Problem: user-space can't access API variables

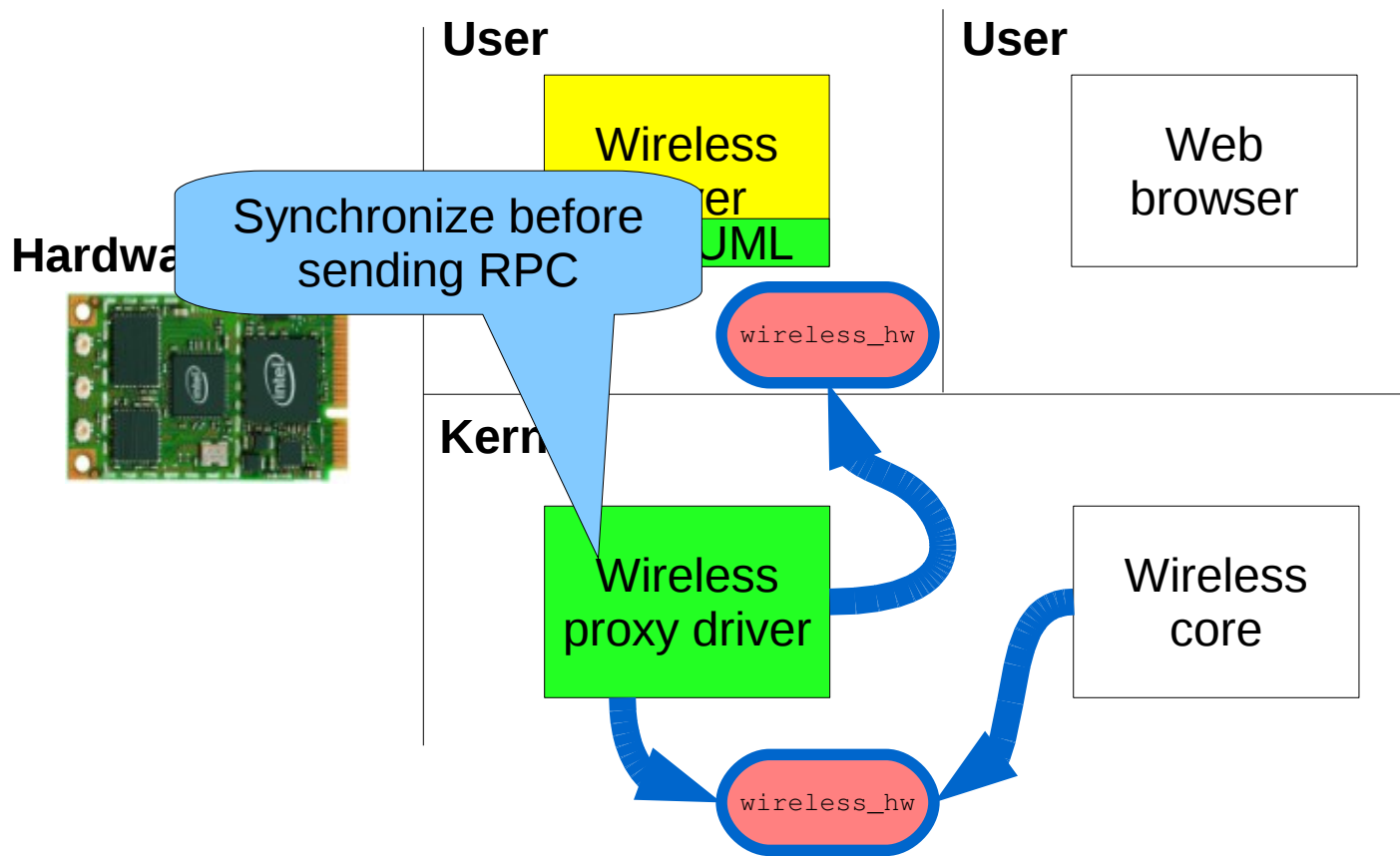
Solution: allocate a shadow copy and synchronize before and after RPCs



Example 3: driver API variables

Problem: user-space can't access API variables

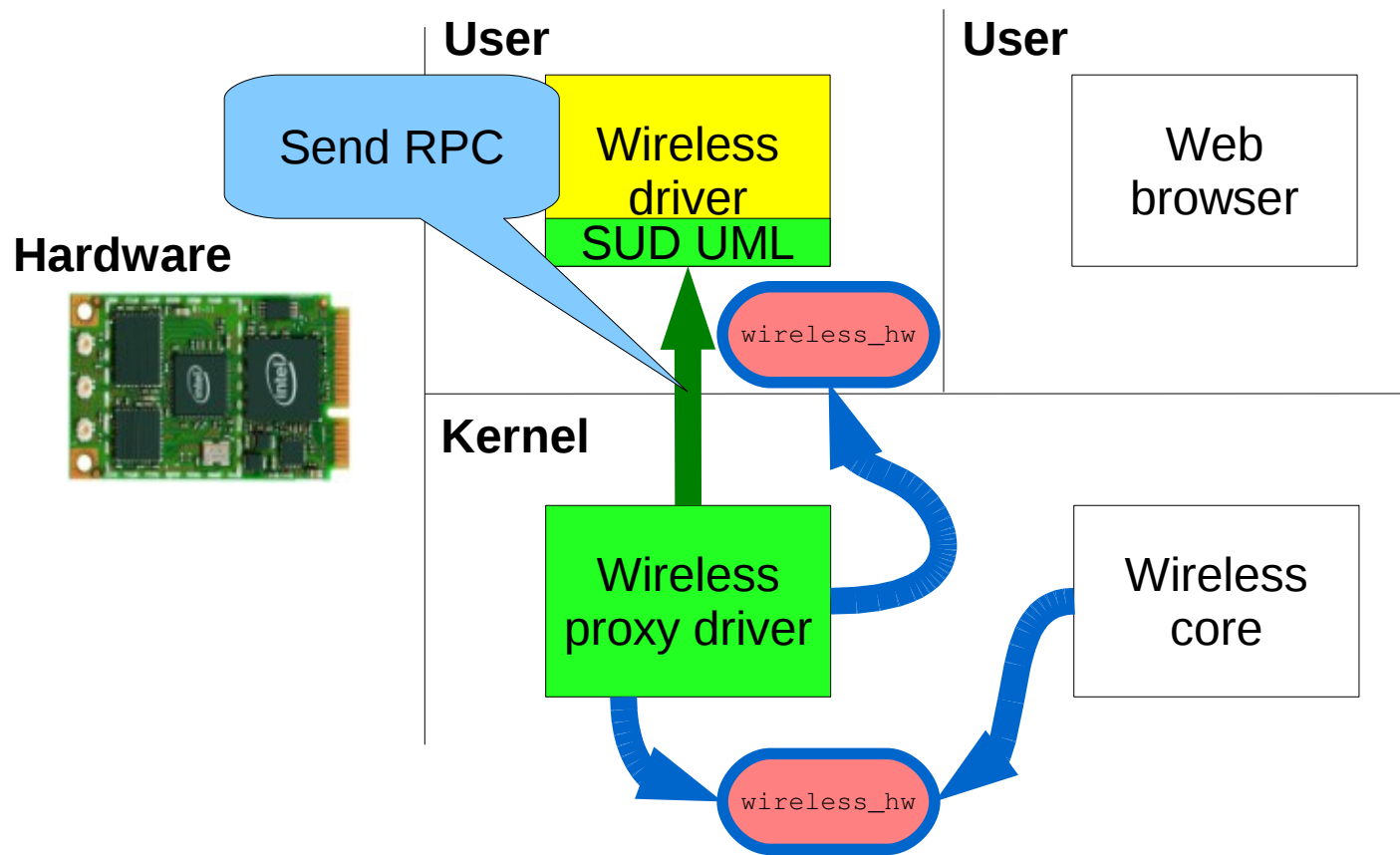
Solution: allocate a shadow copy and synchronize before and after RPCs



Example 3: driver API variables

Problem: user-space can't access API variables

Solution: allocate a shadow copy and synchronize before and after RPCs

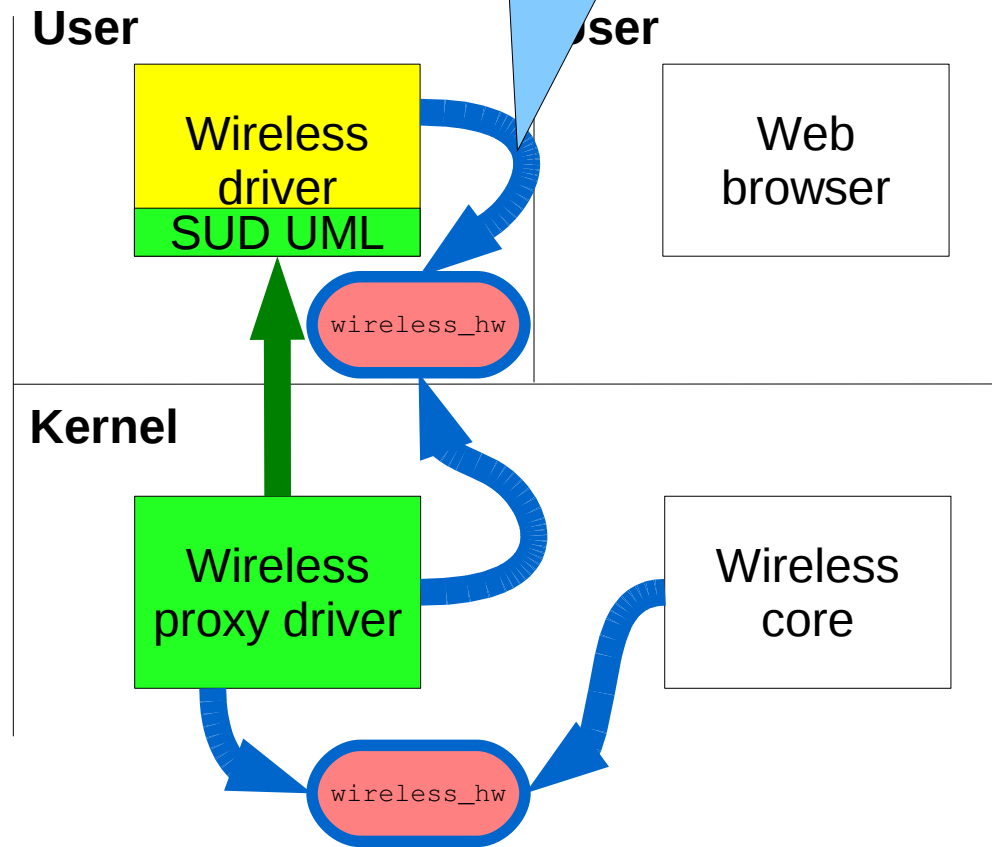


Example 3: driver API variables

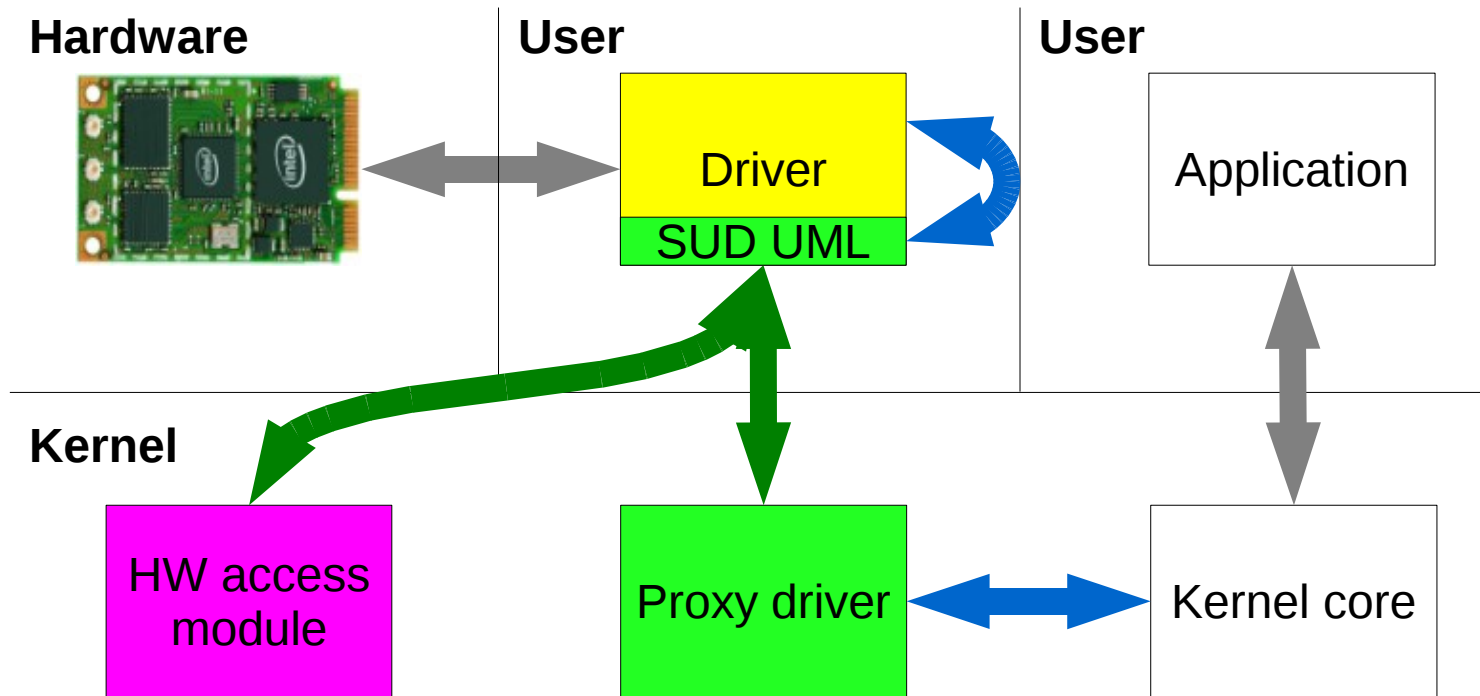
Problem: user-space can't access API variables

Solution: allocate a shadow variable in user-space and synchronize before and after RPCs

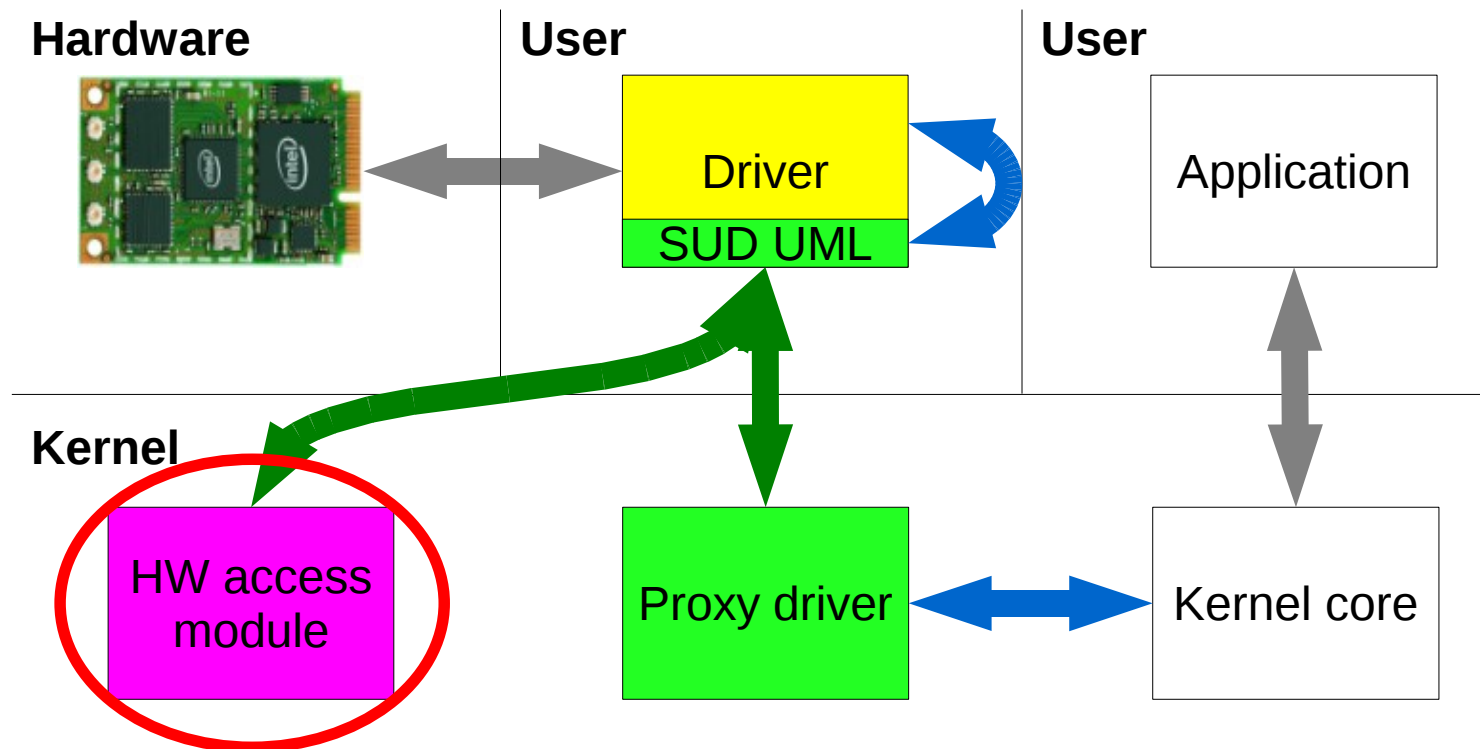
Hardware



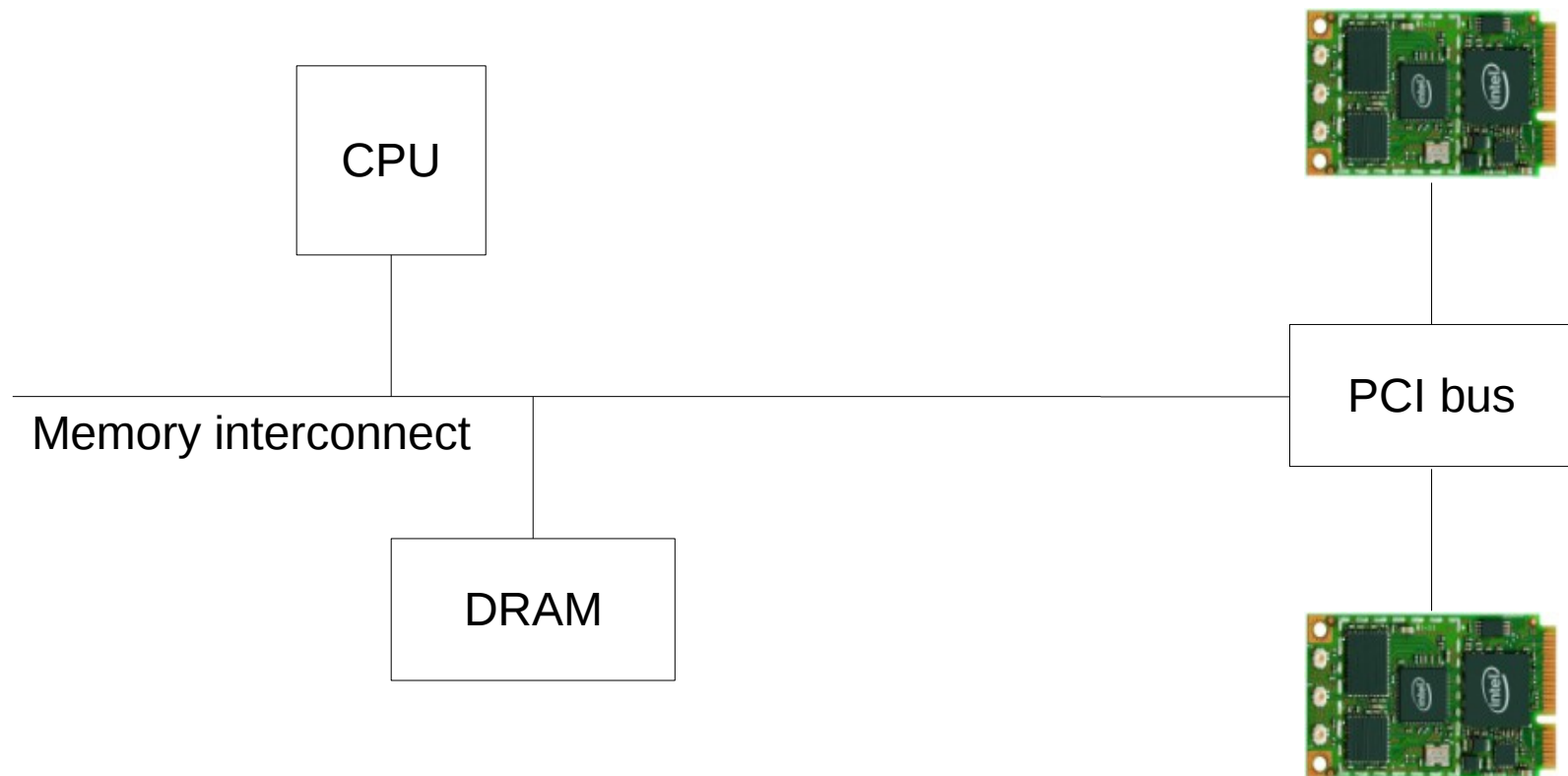
SUD overview



SUD overview

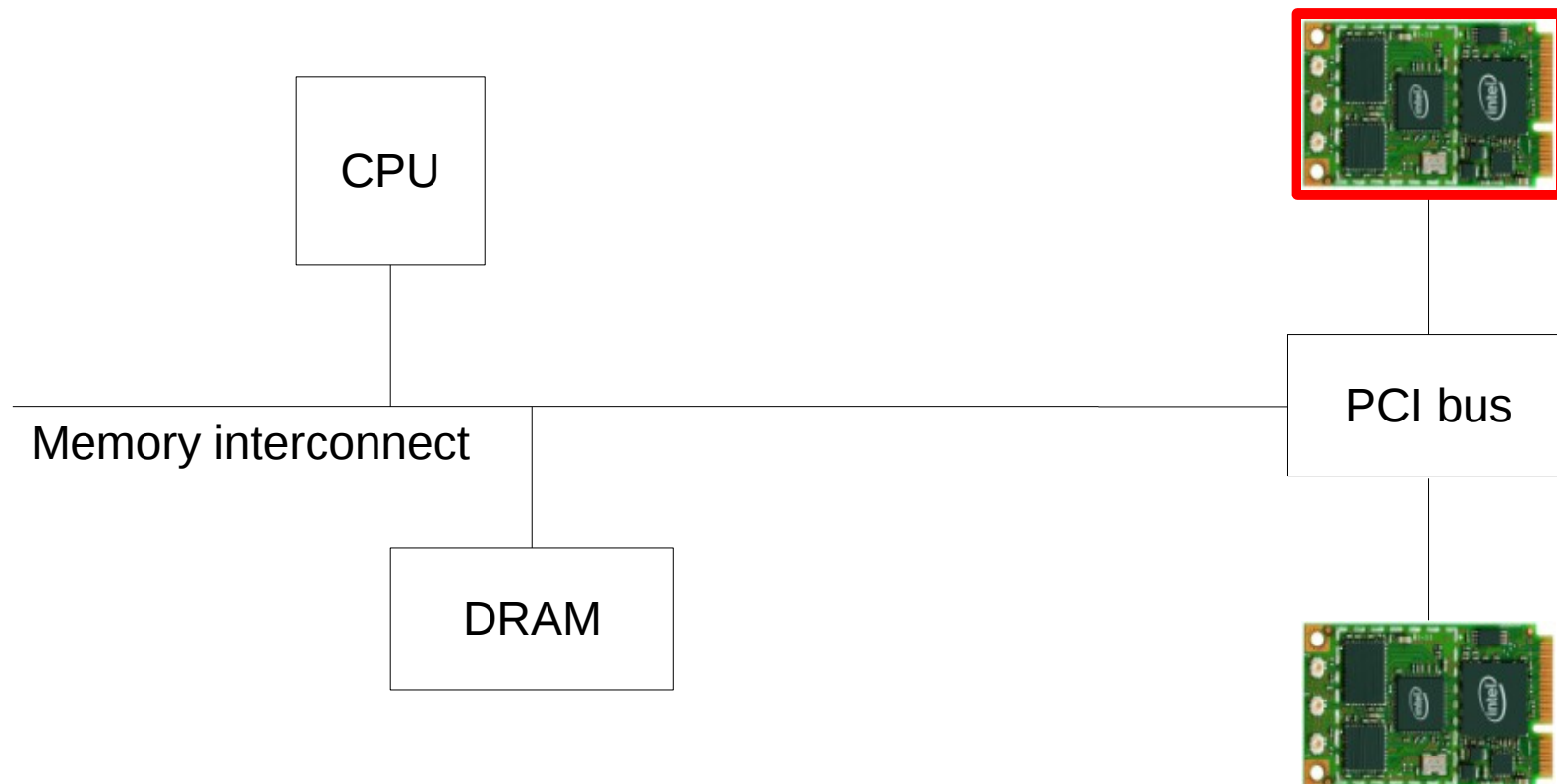


Attacks from hardware



Attacks from hardware

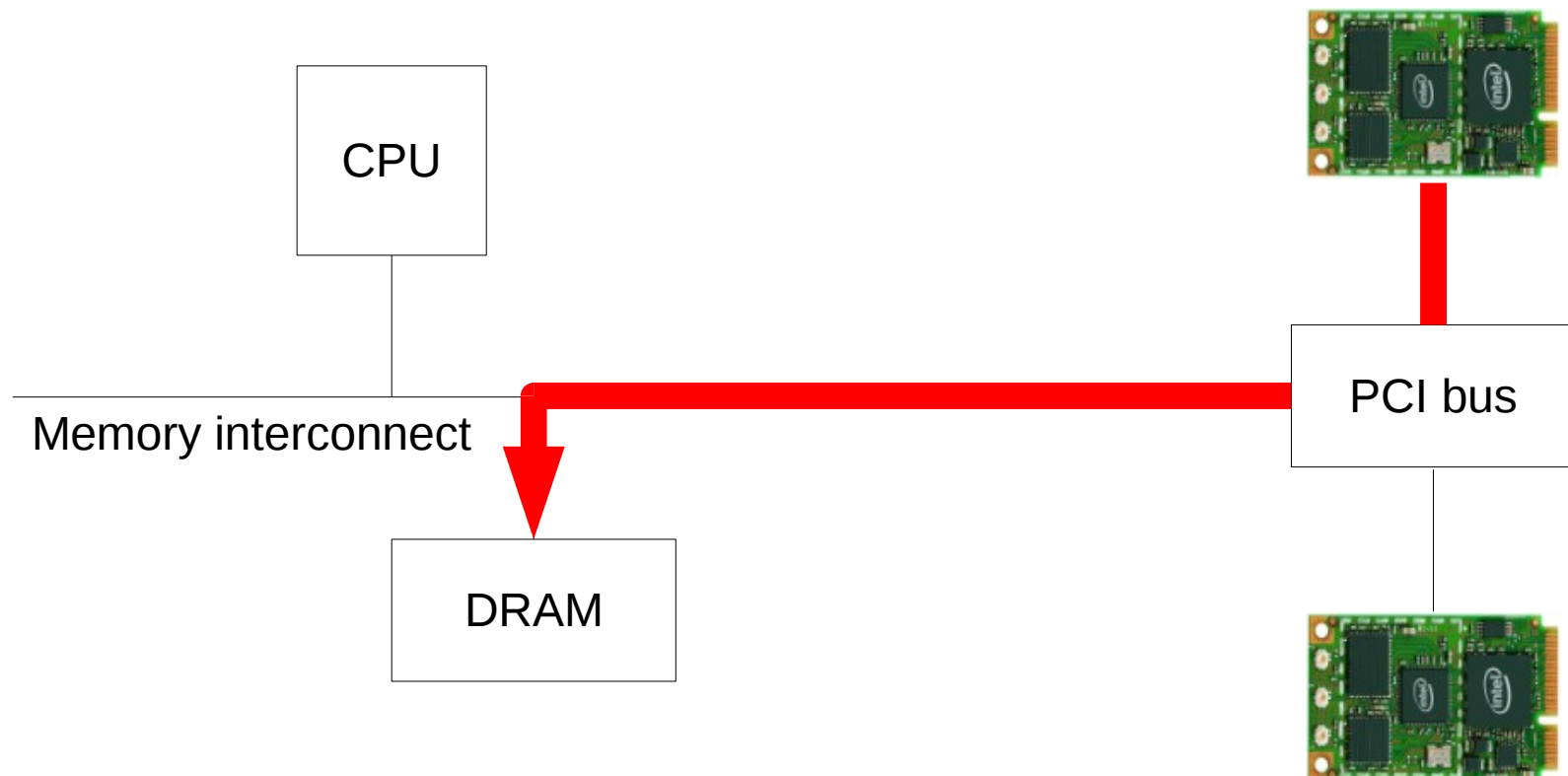
Driver configures the device to execute attacks



Attacks from hardware

Driver configures the device to execute attacks

DMA to DRAM

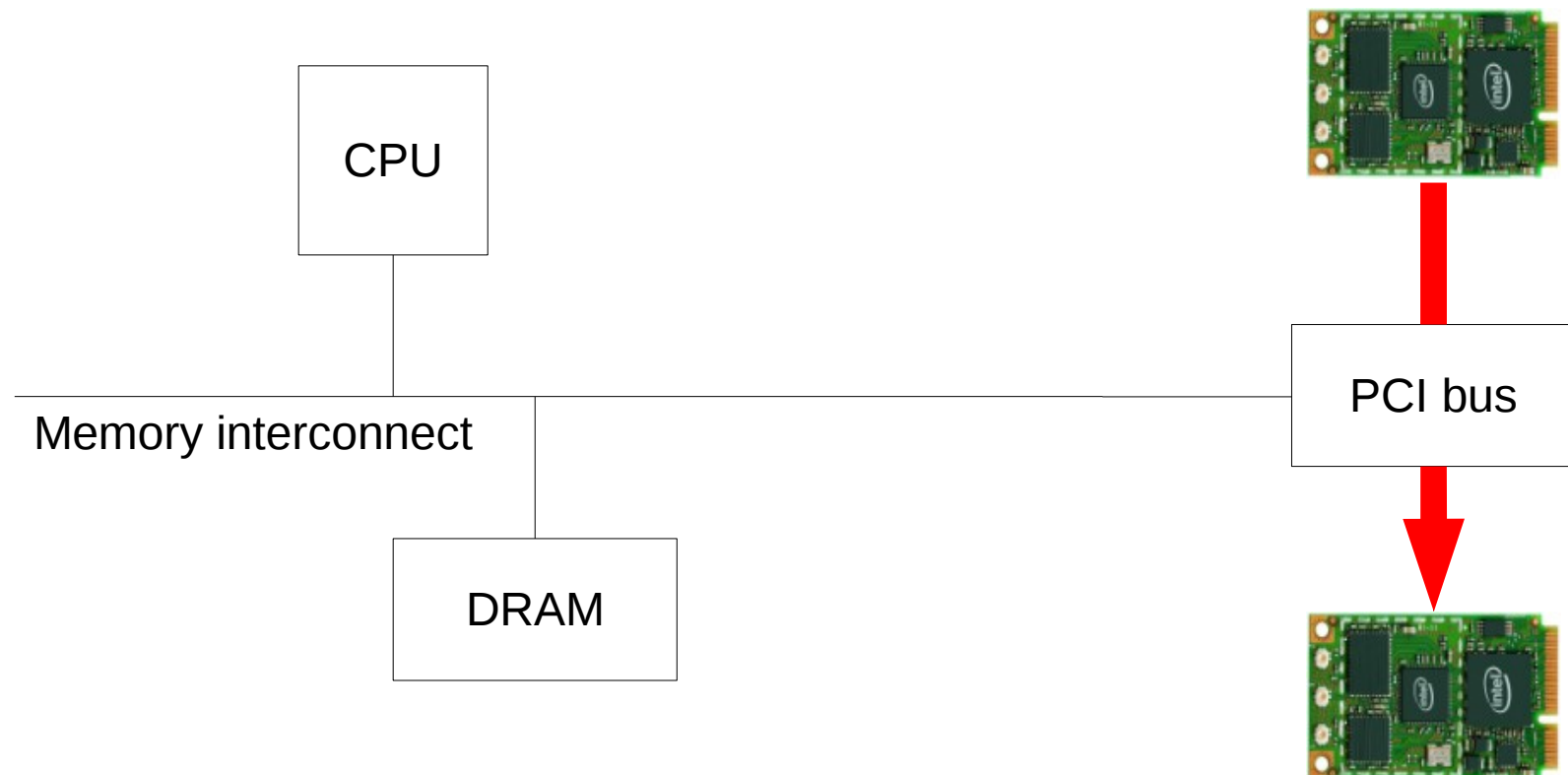


Attacks from hardware

Driver configures the device to execute attacks

DMA to DRAM

Peer-to-peer messages



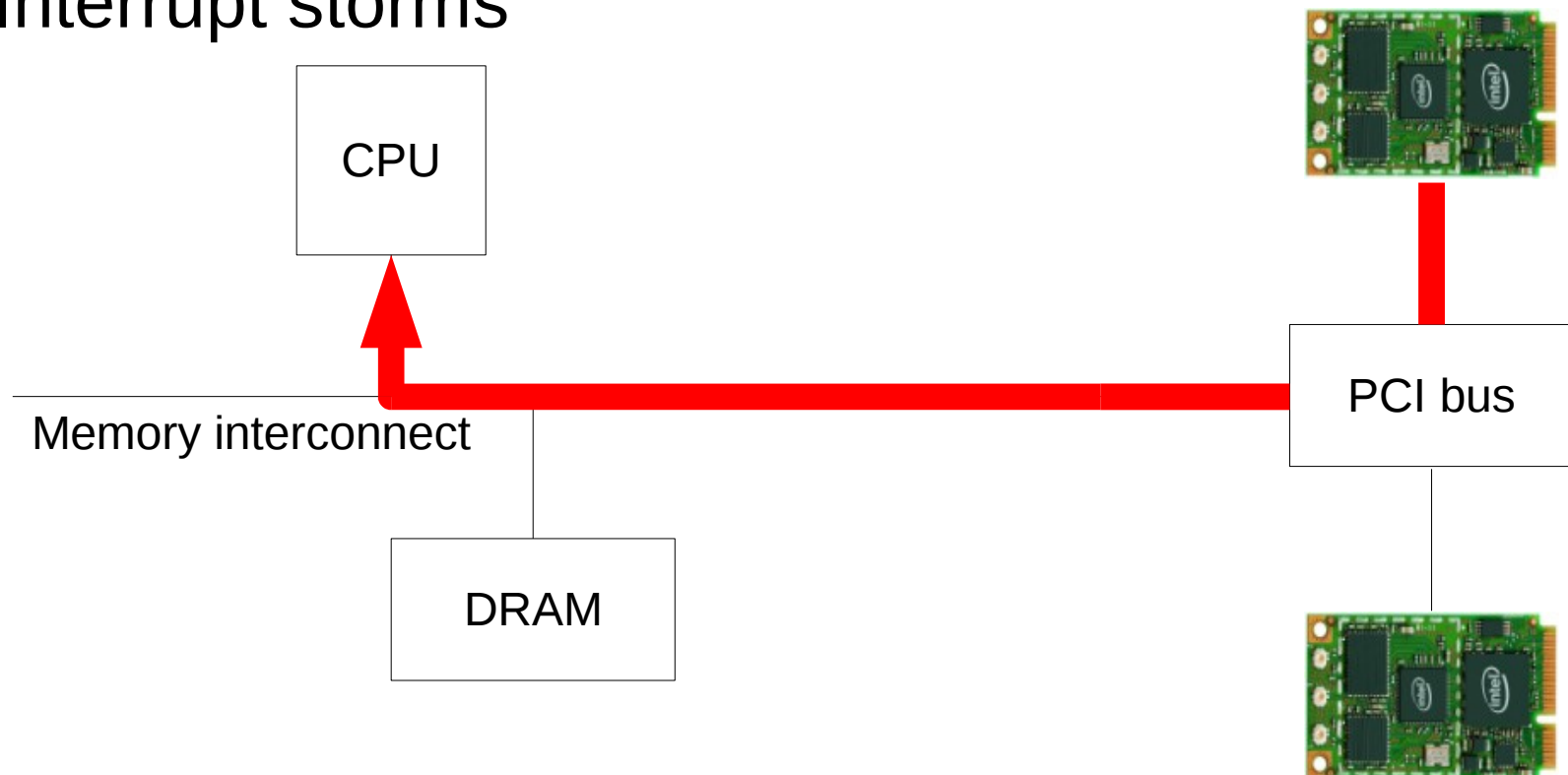
Attacks from hardware

Driver configures the device to execute attacks

DMA to DRAM

Peer-to-peer messages

Interrupt storms



Attacks from hardware

Driver configures the device to execute attacks

- DMA to DRAM

- Peer-to-peer messages

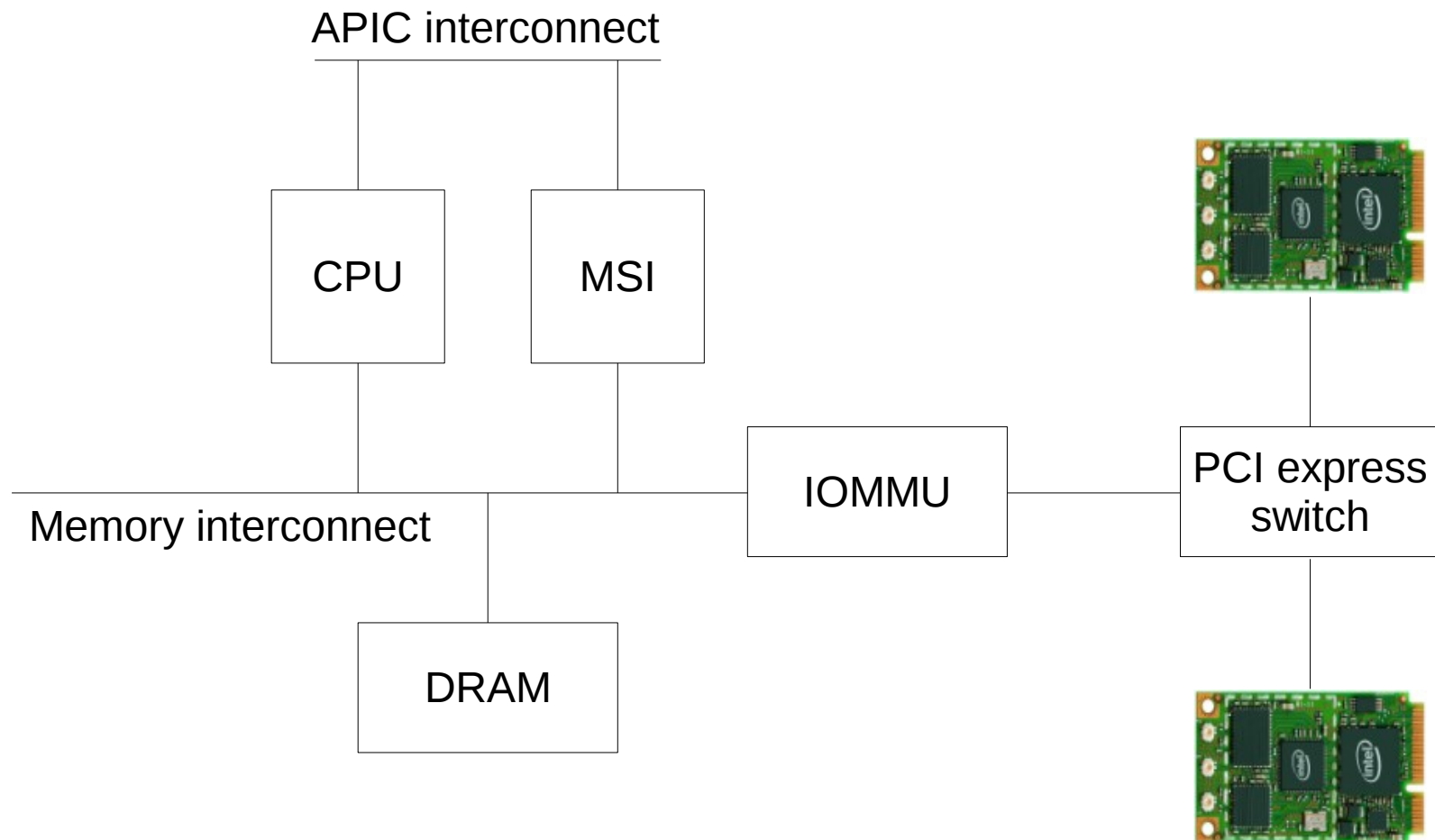
- Interrupt storms

HW access module prevents attacks

- Interposes on driver-device communication

- Uses IO virtualization to provide direct device access

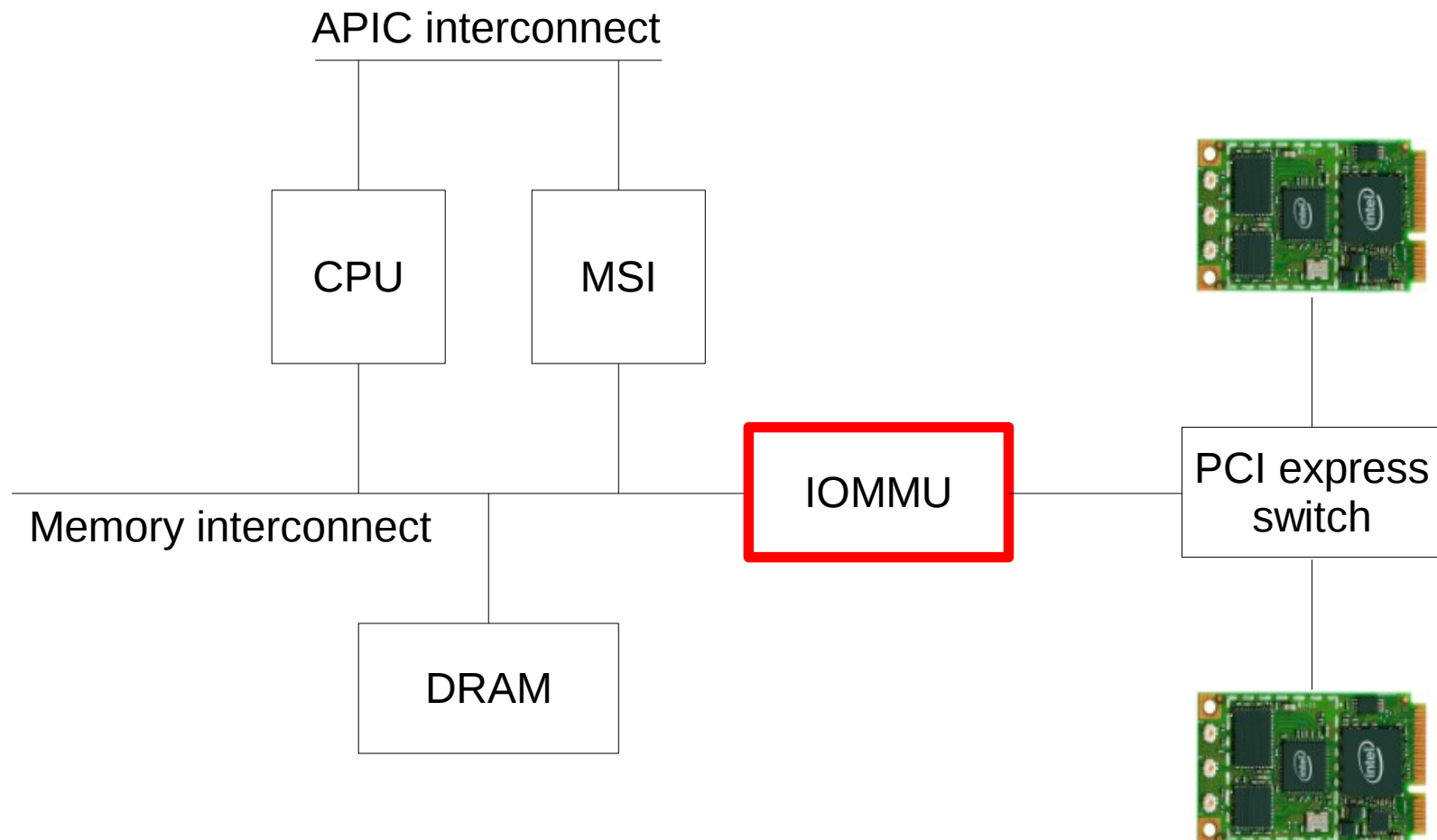
IO virtualization hardware



IO virtualization hardware

Use IOMMU to map DMA buffer pools

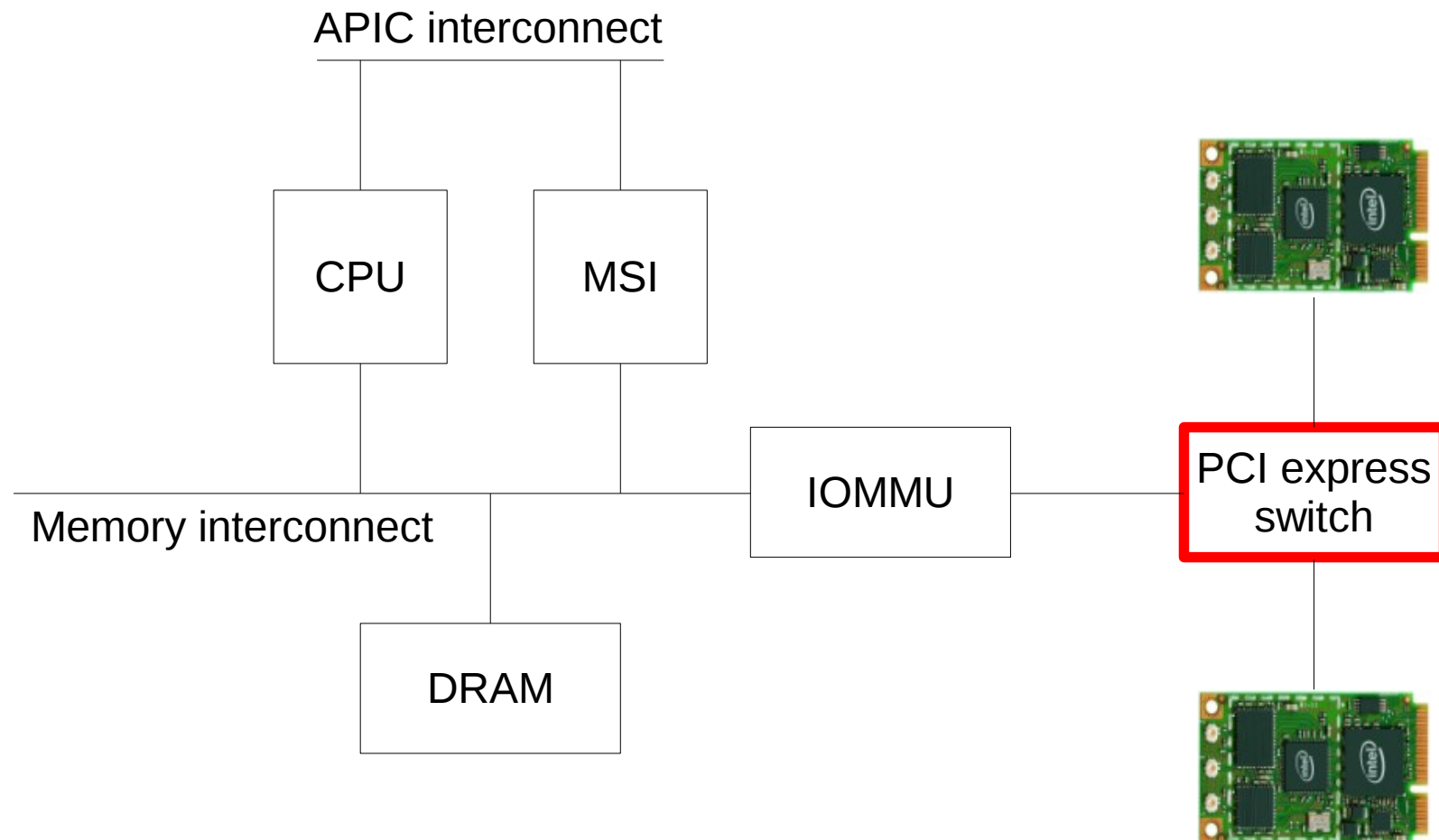
Prevents DMA to DRAM attacks



IO virtualization hardware

Use PCI ACS to prevent peer-to-peer messaging

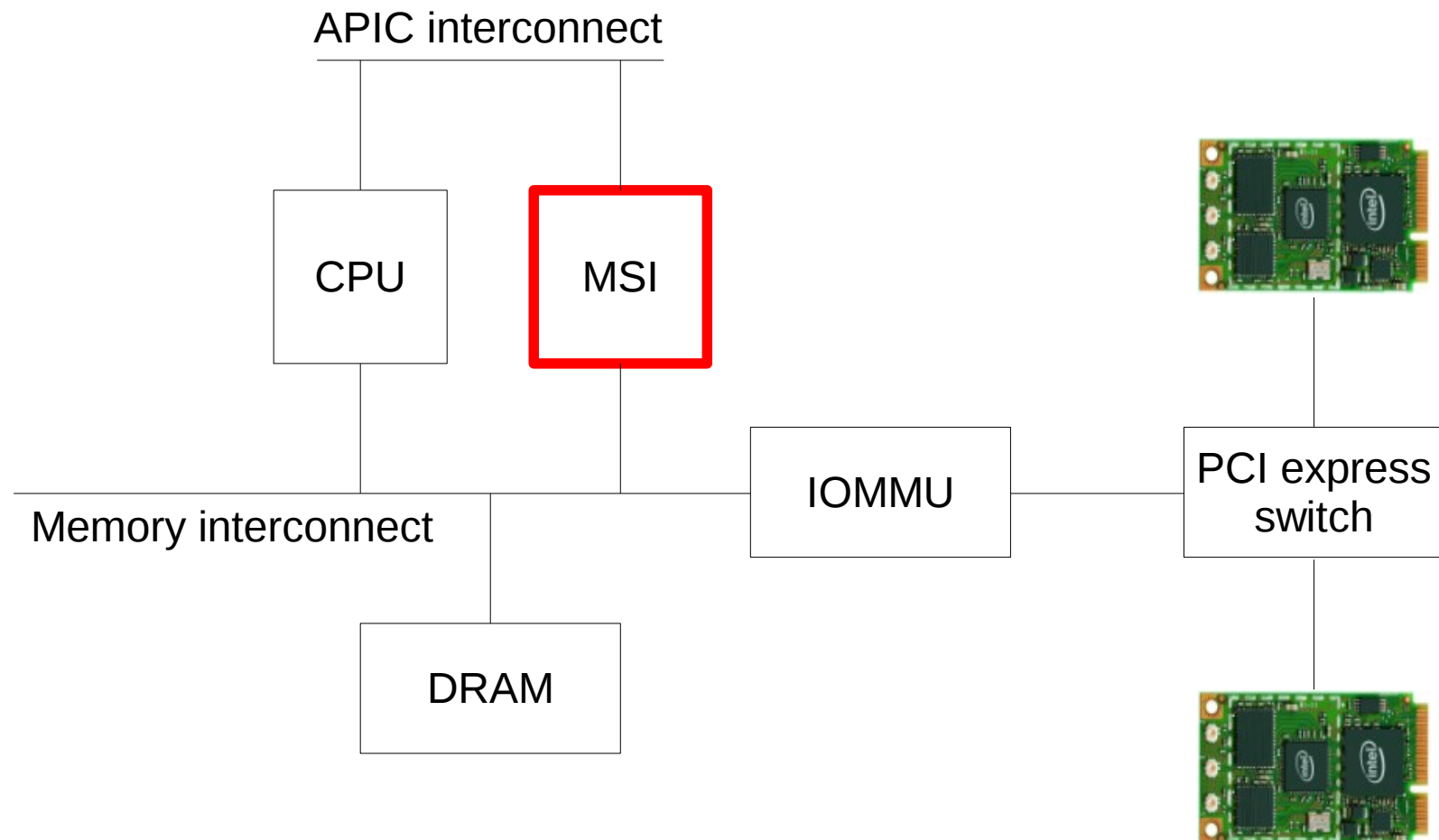
Prevents peer-to-peer attacks



IO virtualization hardware

Use MSI to mask interrupts

Prevents interrupt storms



Interrupt handlers in Linux



MSI

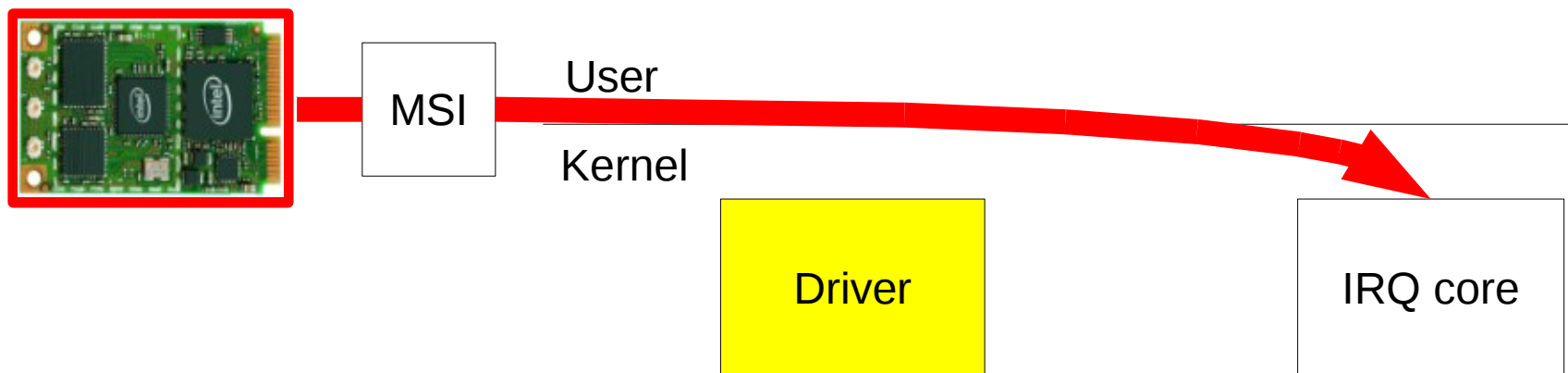
User

Kernel

Driver

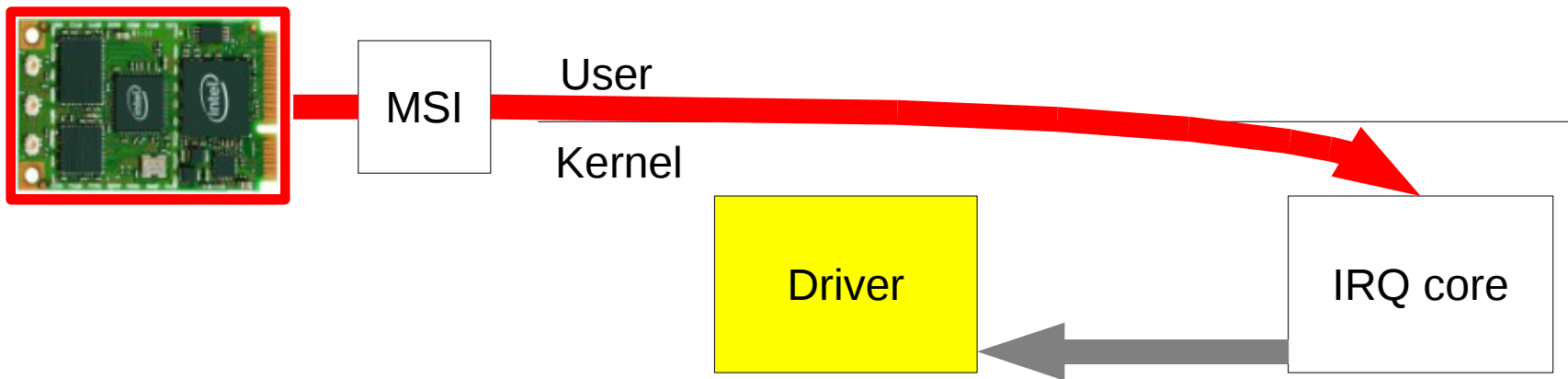
IRQ core

Interrupt handlers in Linux



Interrupt handlers in Linux

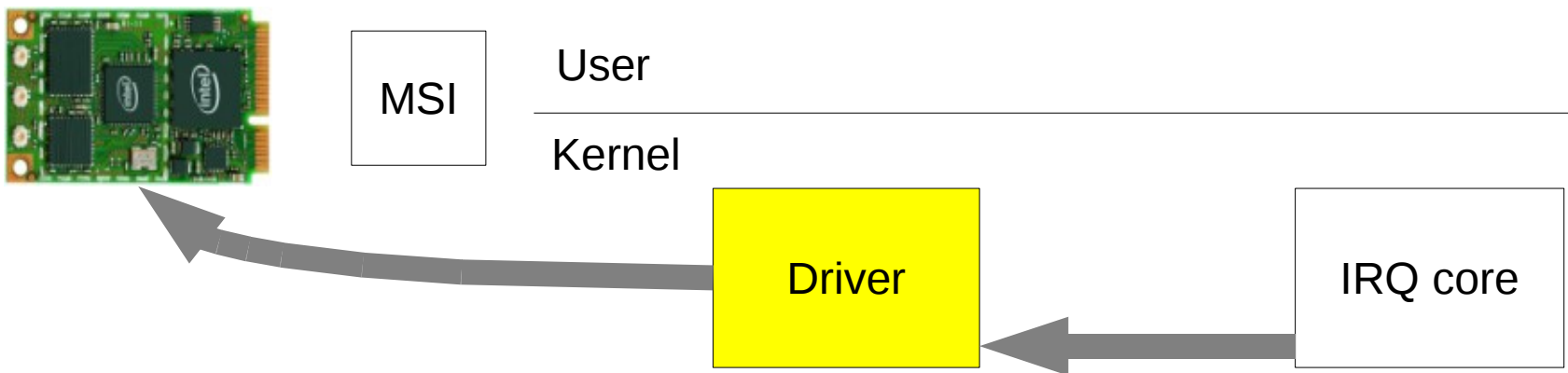
Driver called with IRQs disabled (non-preemptable)



Interrupt handlers in Linux

Kernel calls driver interrupt handler

Driver clears interrupt flag

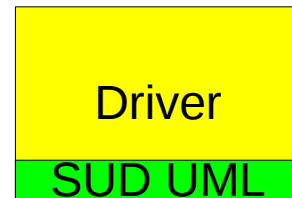


Interrupt handlers with SUD

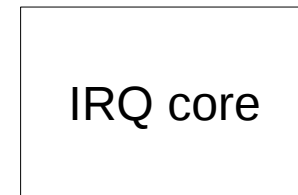
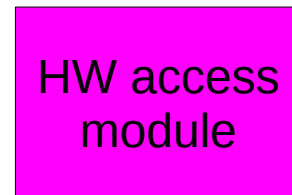


MSI

User



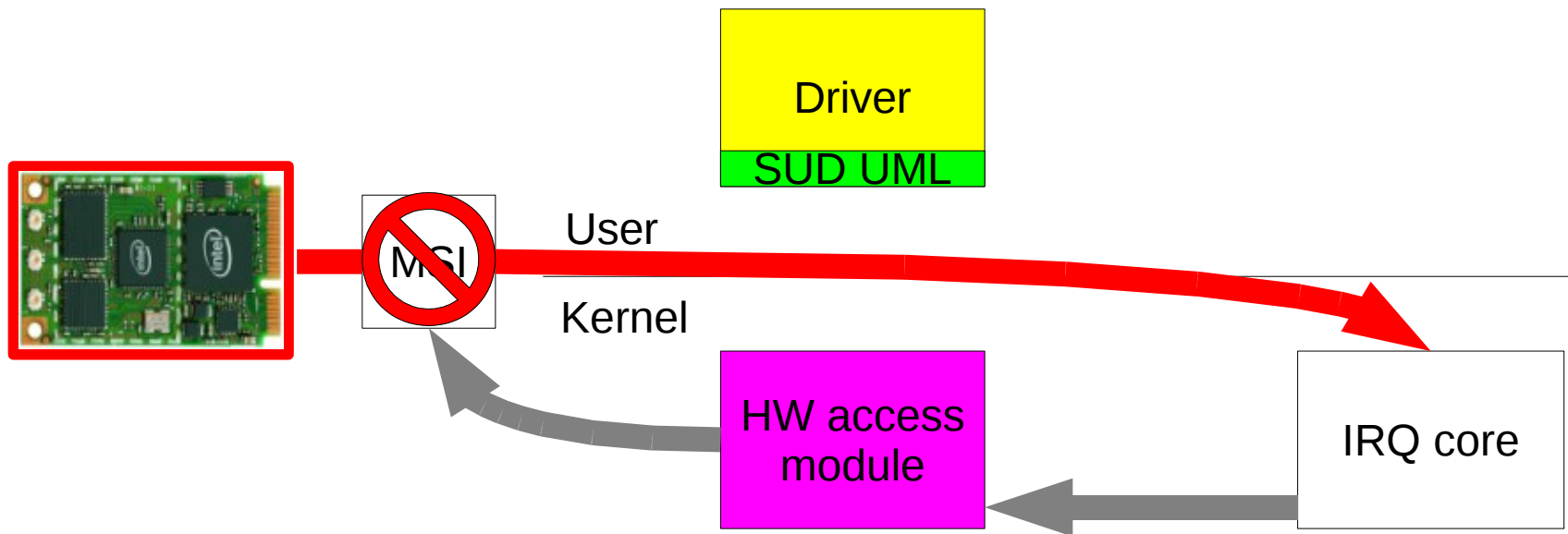
Kernel



Interrupt handlers with SUD

Kernel calls HW access module interrupt handler

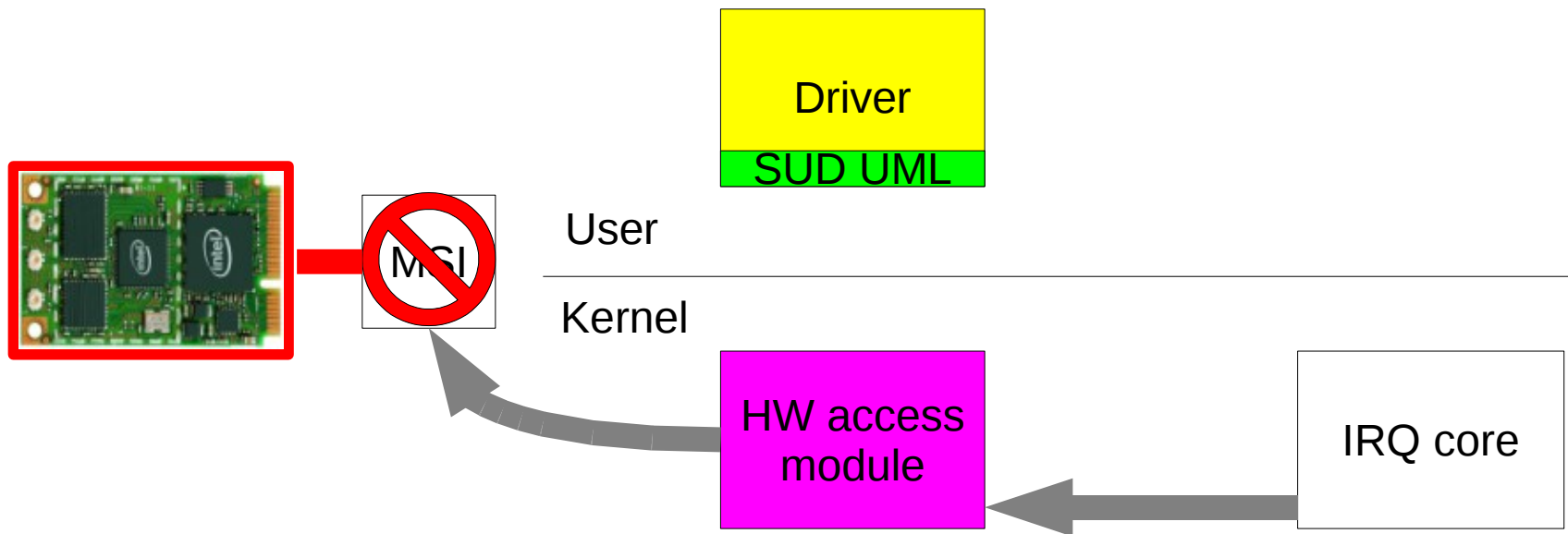
HW access module masks interrupt with MSI



Interrupt handlers with SUD

Kernel calls HW access module interrupt handler

HW access module masks interrupt with MSI

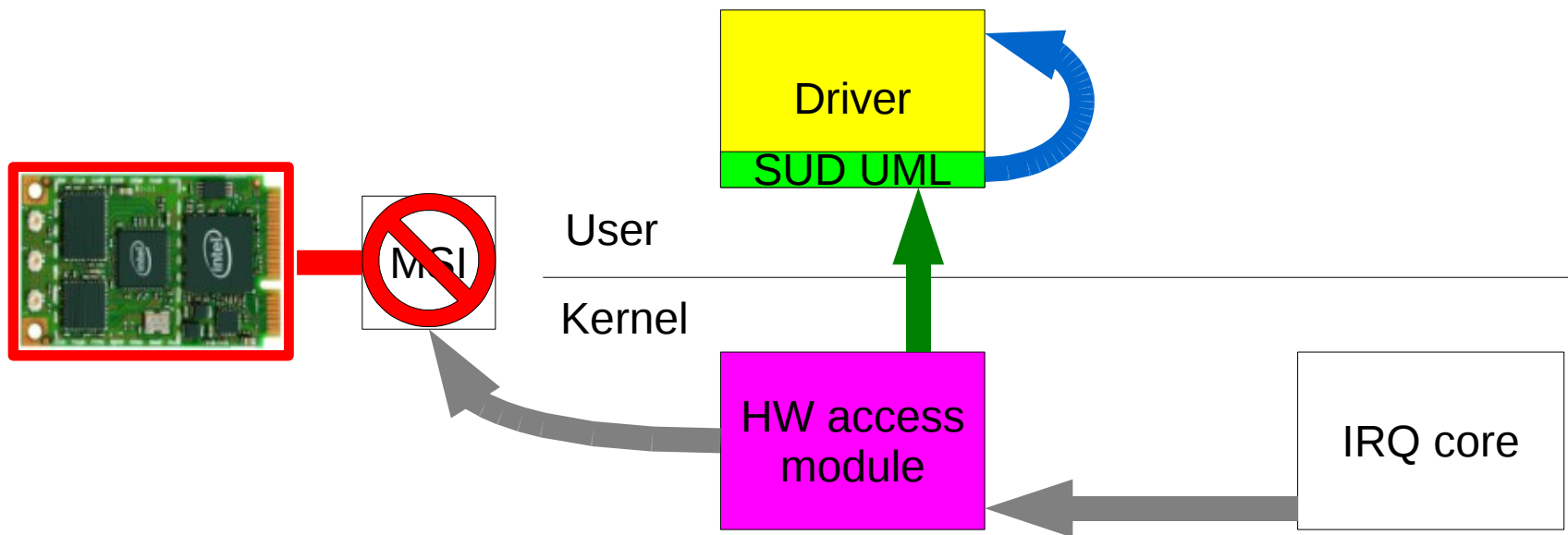


Interrupt handlers with SUD

Kernel calls HW access module interrupt handler

HW access module masks interrupt with MSI

Asynchronous RPC to driver



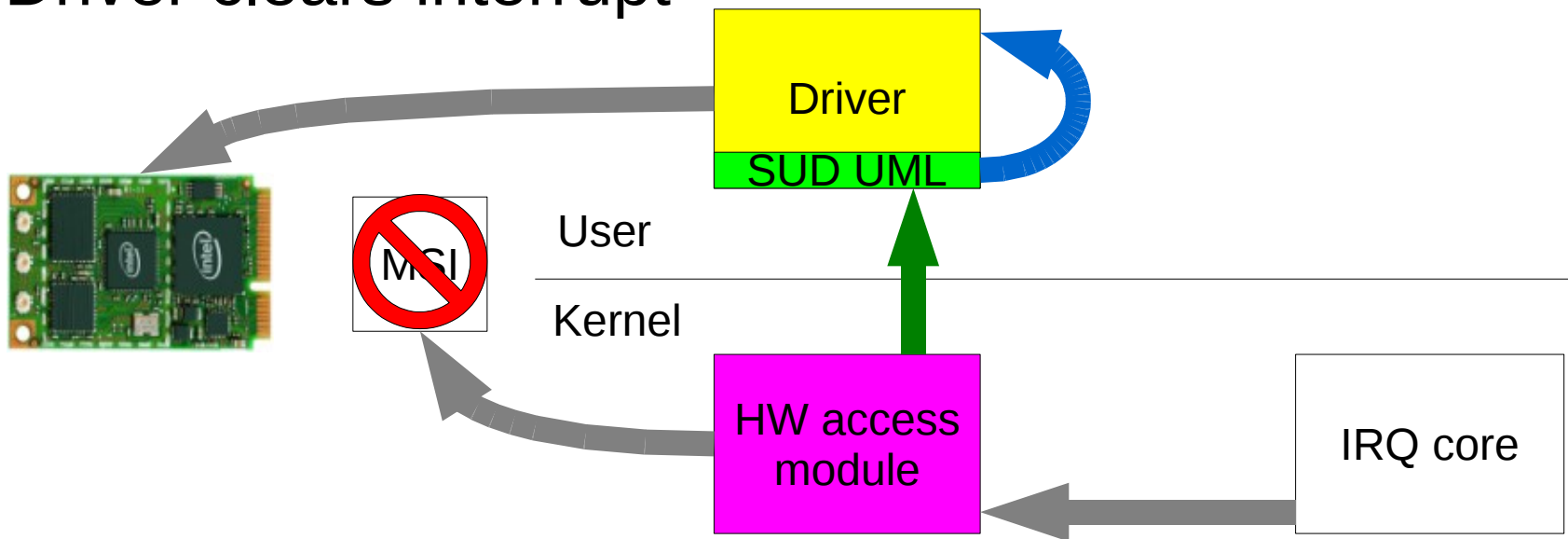
Interrupt handlers with SUD

Kernel calls HW access module interrupt handler

HW access module masks interrupt with MSI

Asynchronous RPC to driver

Driver clears interrupt



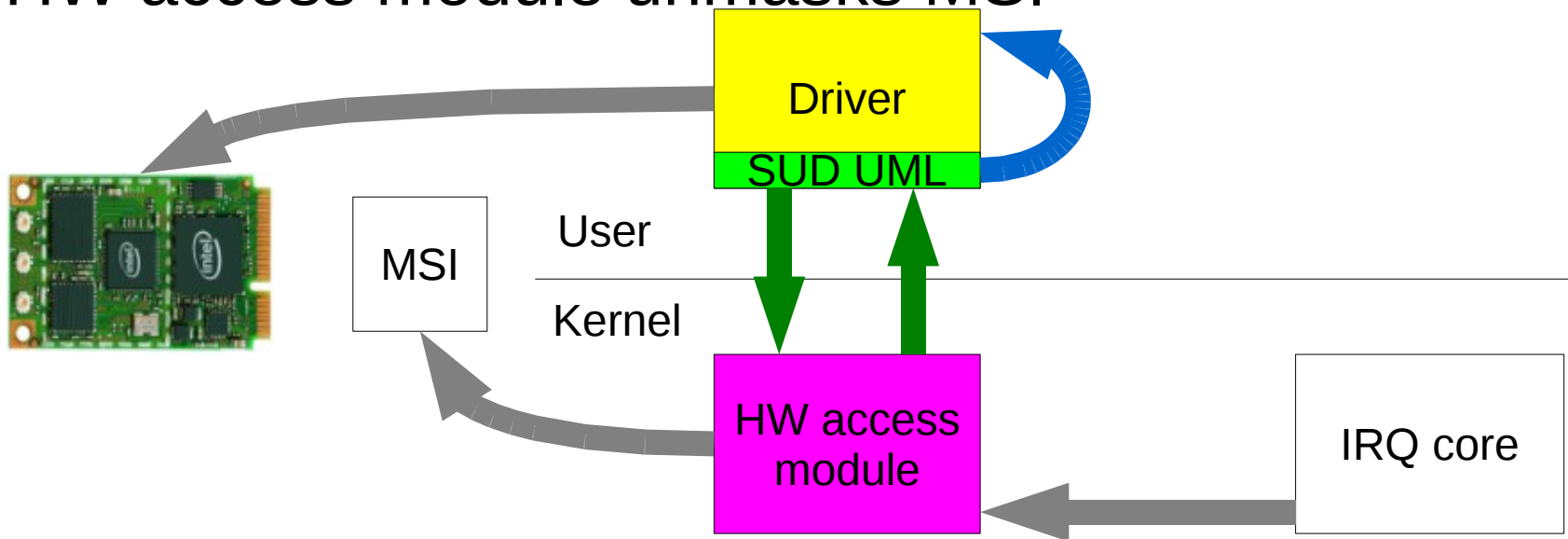
Interrupt handlers with SUD

HW access module masks interrupt with MSI

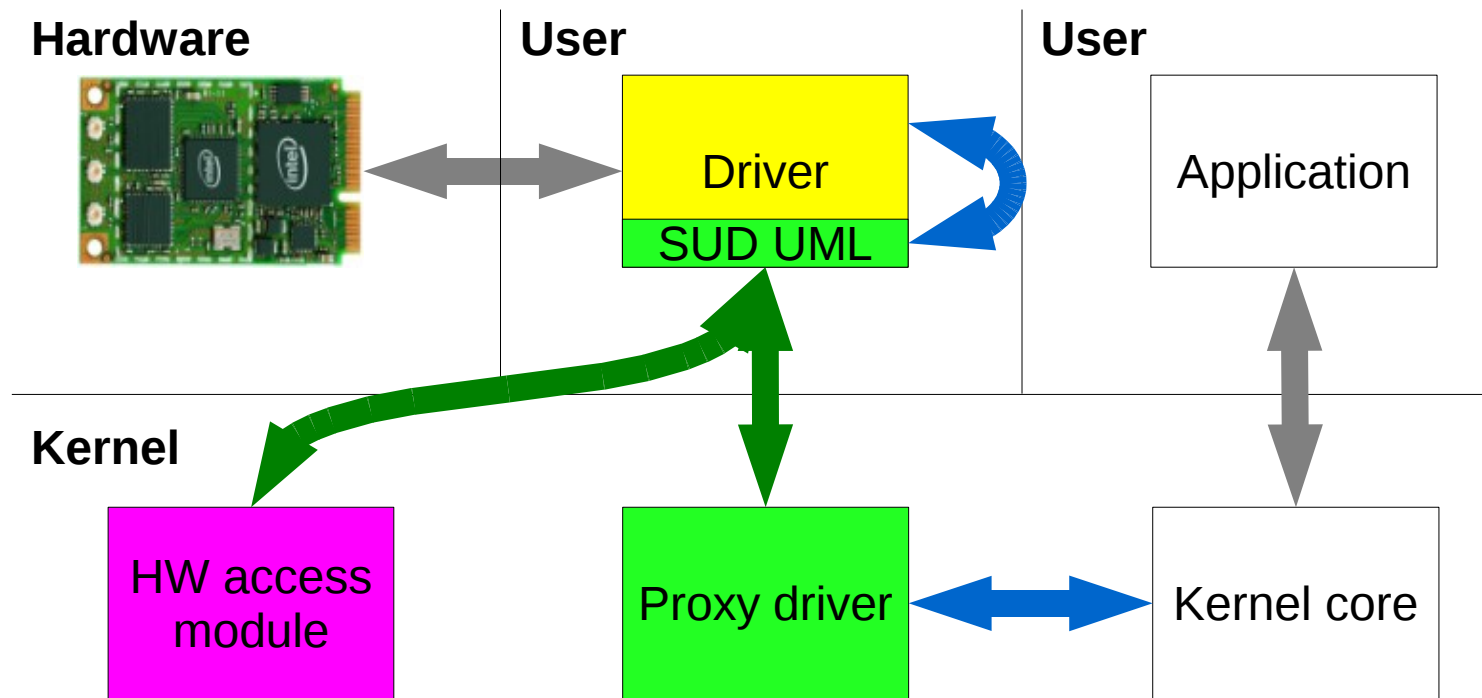
Asynchronous RPC to driver

Driver clears interrupt

HW access module un.masks MSI



SUD overview



Prototype of SUD

Trusted code	Lines of code
PCI access module	2800
Ethernet proxy driver	300
Wireless proxy driver	600
Audio proxy driver	550

Untrusted code	Lines of code
User-mode runtime	5000
Drivers	5000 – 50,000 (each)

Supports all Ethernet, wireless, USB, audio drivers

Tested: e1000e, ne2k-pci, iwlgagn, snd_hda_intel, ehci_hcd, uhci_hcd, ...

Prototype of SUD

Trusted code	Lines of code
PCI access module	2800
Ethernet proxy driver	300
Wireless proxy driver	600
Audio proxy driver	550

Untrusted code	Lines of code
User-mode runtime	5000
Drivers	5000 – 50,000 (each)

Supports all Ethernet, wireless, USB, audio drivers

Tested: e1000e, ne2k-pci, iwlgagn, snd_hda_intel, ehci_hcd, uhci_hcd, ...

Prototype of SUD

Trusted code	Lines of code
PCI access module	2800
Ethernet proxy driver	300
Wireless proxy driver	600
Audio proxy driver	550

Untrusted code	Lines of code
User-mode runtime	5000
Drivers	5000 – 50,000 (each)

Supports all Ethernet, wireless, USB, audio drivers

Tested: e1000e, ne2k-pci, iwlgagn, snd_hda_intel, ehci_hcd, uhci_hcd, ...

Performance

For most devices, does not matter

Printers, cameras, ...

Stress-test: e1000e gigabit network card

Requires high throughput

Requires low latency

Many device driver interactions

Test machine: 1.4GHz dual core Thinkpad

Performance questions?

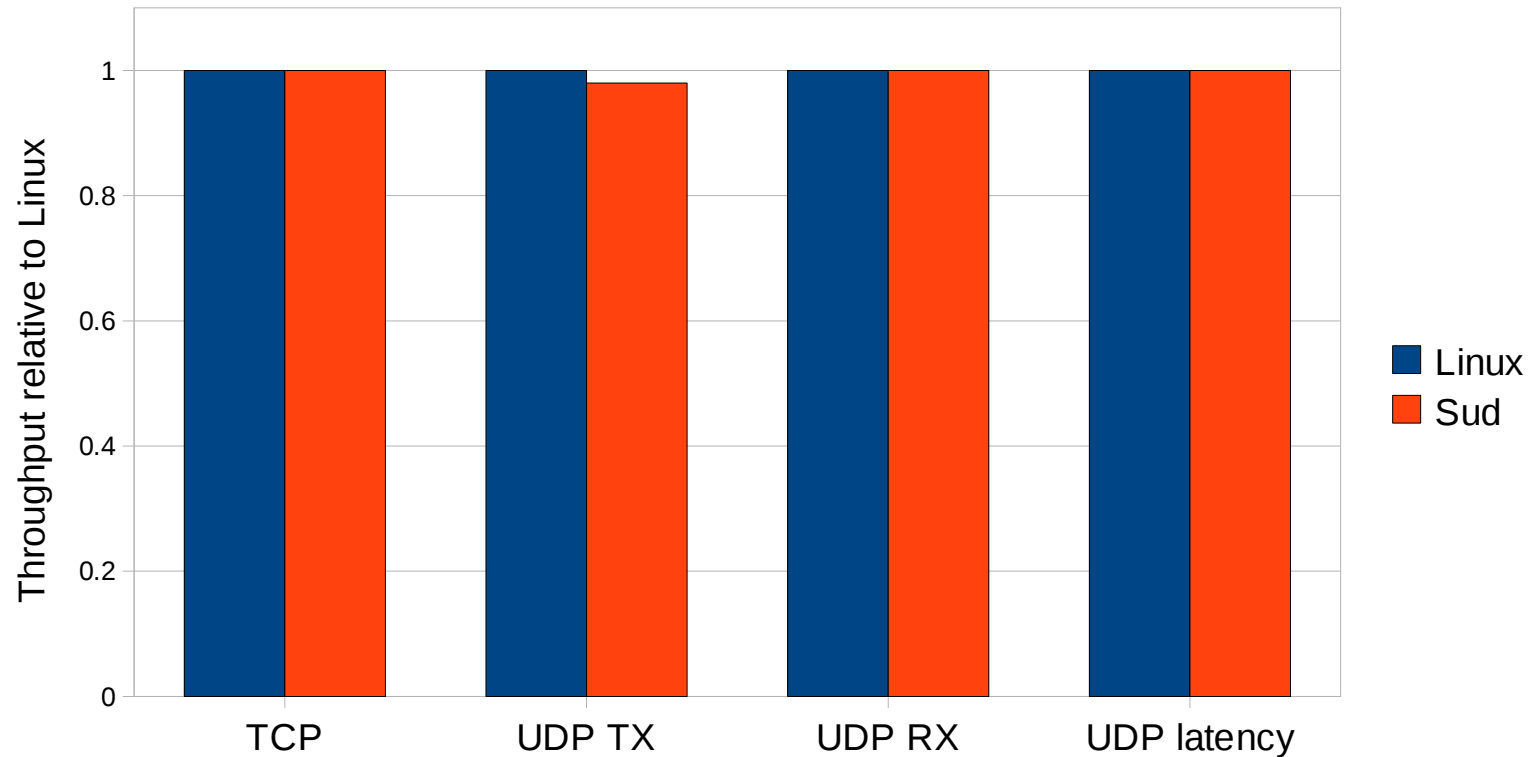
What performance does SUD get?

Network throughput, latency

How much does it cost?

CPU cycles

SUD achieves same device performance

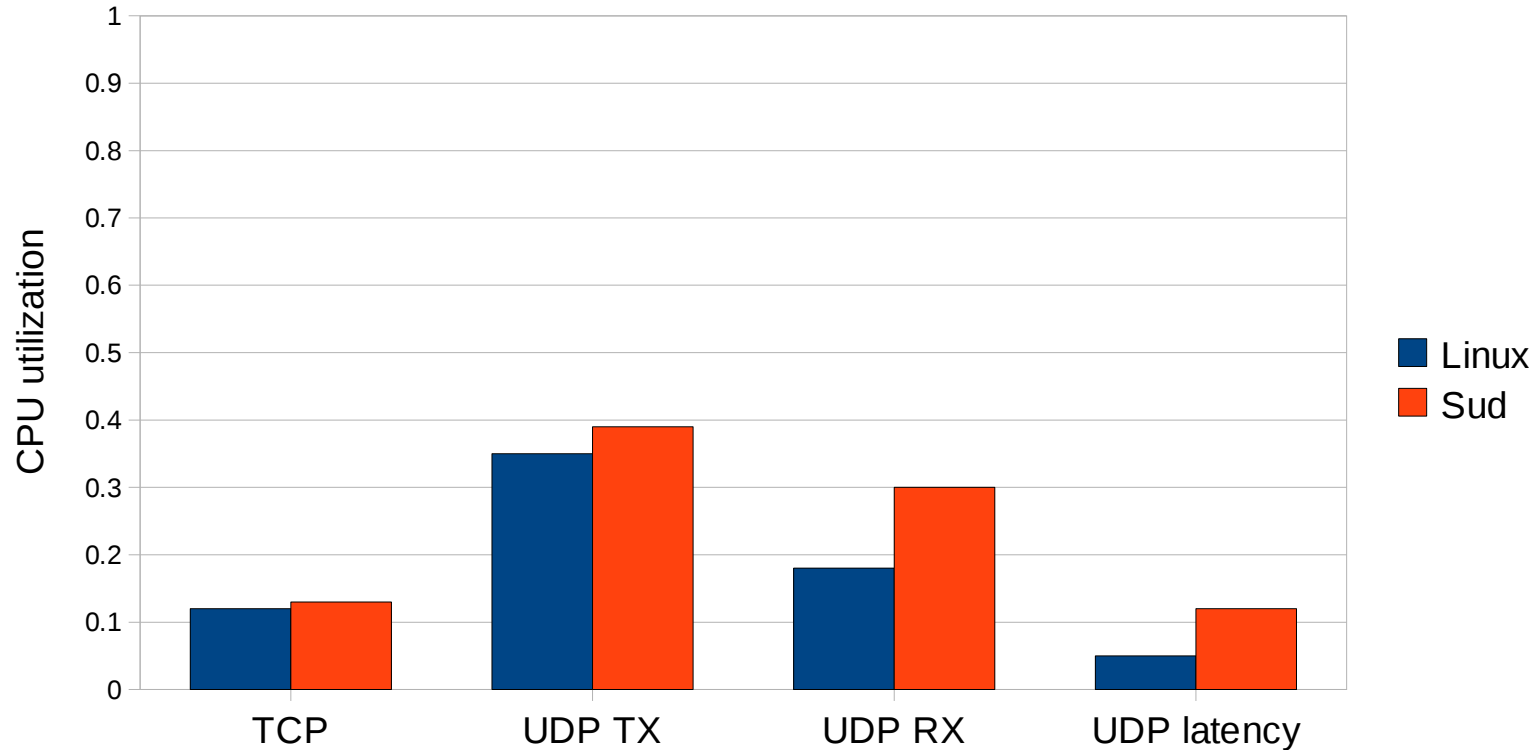


Normalized throughput relative to Linux

TCP: streaming (950 Mbps in both cases)

UDP: one-byte-data packets

CPU cost is low



SUD overhead: user-kernel switch, TLB misses

Overheads not significant for many workloads
(packets larger than min. packet size)

Future directions

Explore hierarchical untrusted device drivers

PCI bus → SATA controller → SATA disk → ...

Explore giving apps direct hardware access

Safe HW access for network analyzer, X server, ...

Performance analysis and optimizations

SUD specific device drivers, super pages, ...

Related work

Mircokernels (Minix, L4, ...)

Simple drivers, driver API designed for user-space

Nooks, microdrivers

Handles common bugs, many changes to kernel

Languages (e.g. Termite), source code analysis

Complimentary to user-space drivers

No need for new OS or language

Summary

Driver bugs lead to system crashes or exploits

SUD protects Linux from malicious drivers using proxy drivers and IO virtualization HW

- Runs unmodified Linux device drivers

- High performance, low overheads

- Few modifications to Linux kernel

Security evaluation

Manually constructed potential attacks

Memory corruption, arbitrary upcall responses,
not responding at all, arbitrary DMA, ...

Relied on security heavily during development

SUD caught all bugs in user-mode driver framework

No crashes / reboots required to develop drivers

Ideal, but not done: red-team evaluation?