CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules

Thomas Filipiuk, **Nick Wanninger**, Nadharm Dhiantravan, Carson H Surmeier, Alex Bernat, Peter Dinda

SC23 Denver, CO | i am hpc.

CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules

Thomas Filipiuk[1], **Nick Wanninger**[1], Nadharm Dhiantravan[1], Carson H Surmeier[1], Alex Bernat[2], Peter Dinda[1]

Northwestern University[1], Harvard University[2]

Applications and Runtimes can be accelerated with custom extensions to the operating system


2016 IEEE International Parallel and Distributed Processing Symposium

On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel

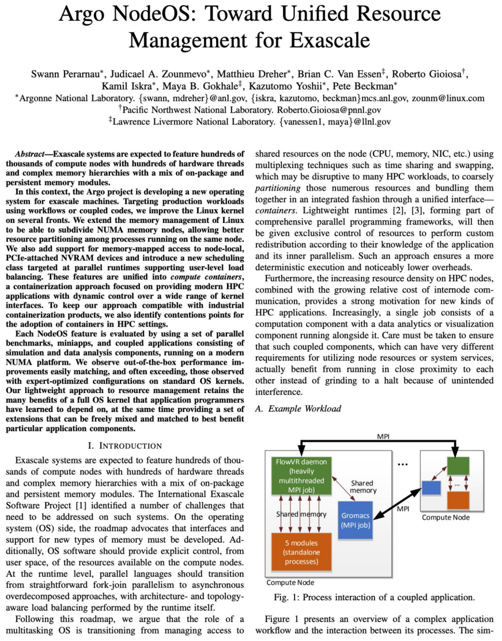
Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura†, Tomoki Shirasawa‡ and Yutaka Ishikawa
RIKEN Advanced Institute for Computational Science, JAPAN
†Hitachi Solutions, Ltd., JAPAN
‡Hitachi Solutions East Japan, Ltd., JAPAN

IHK/MCKernel


Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing

John Lange*, Kevin Pedretti†, Trammell Hudson†, Peter Dinda*, Zheng Cui‡, Lei Xia*, Patrick Bridges‡, Andy Gocke*, Steven Jaconette*, Mike Levenhagen†, and Ron Brightwell†
* Northwestern University, Department of Electrical Engineering and Computer Science
† Sandia National Laboratories, Scalable System Software Department
‡ University of New Mexico, Department of Computer Science

Palacios + Kitten


Argo NodeOS: Toward Unified Resource Management for Exascale

Swann Perarnau*, Judicael A. Zounmevo*, Matthieu Dreher*, Brian C. Van Essen‡, Roberto Gioiosa†, Kamil Iskra*, Maya B. Gokhale‡, Kazutomo Yoshii*, Pete Beckman*
*Argonne National Laboratory
†Pacific Northwest National Laboratory
‡Lawrence Livermore National Laboratory

Argo NodeOS


2017 IEEE International Conference on Autonomic Computing

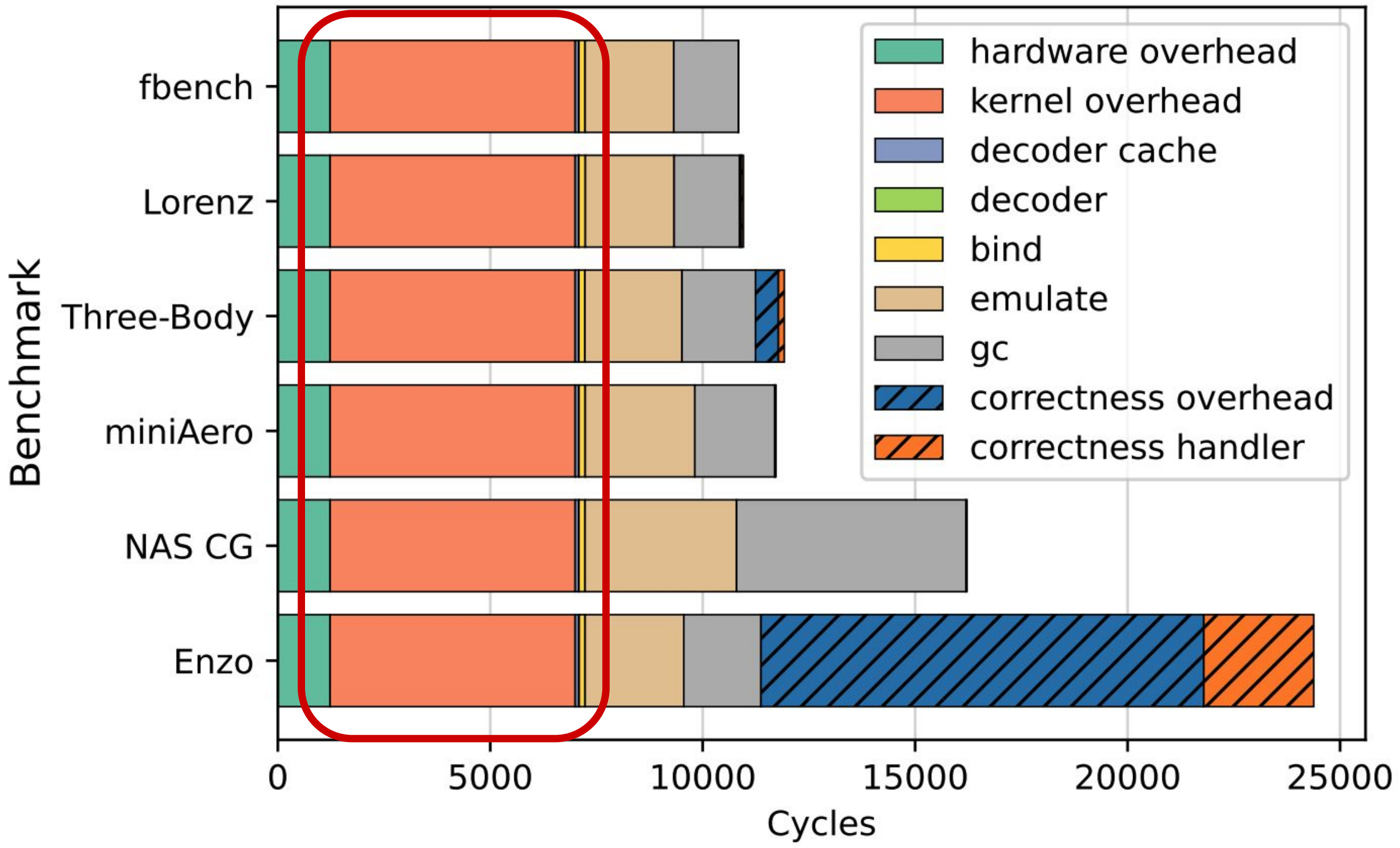
Multiverse: Easy Conversion of Runtime Systems into OS Kernels via Automatic Hybridization

Kyle C. Hale, Department of Computer Science, Illinois Institute of Technology
Conor Hetland, Peter Dinda, Department of Electrical Engineering and Computer Science, Northwestern University

Nautilus/Multiverse

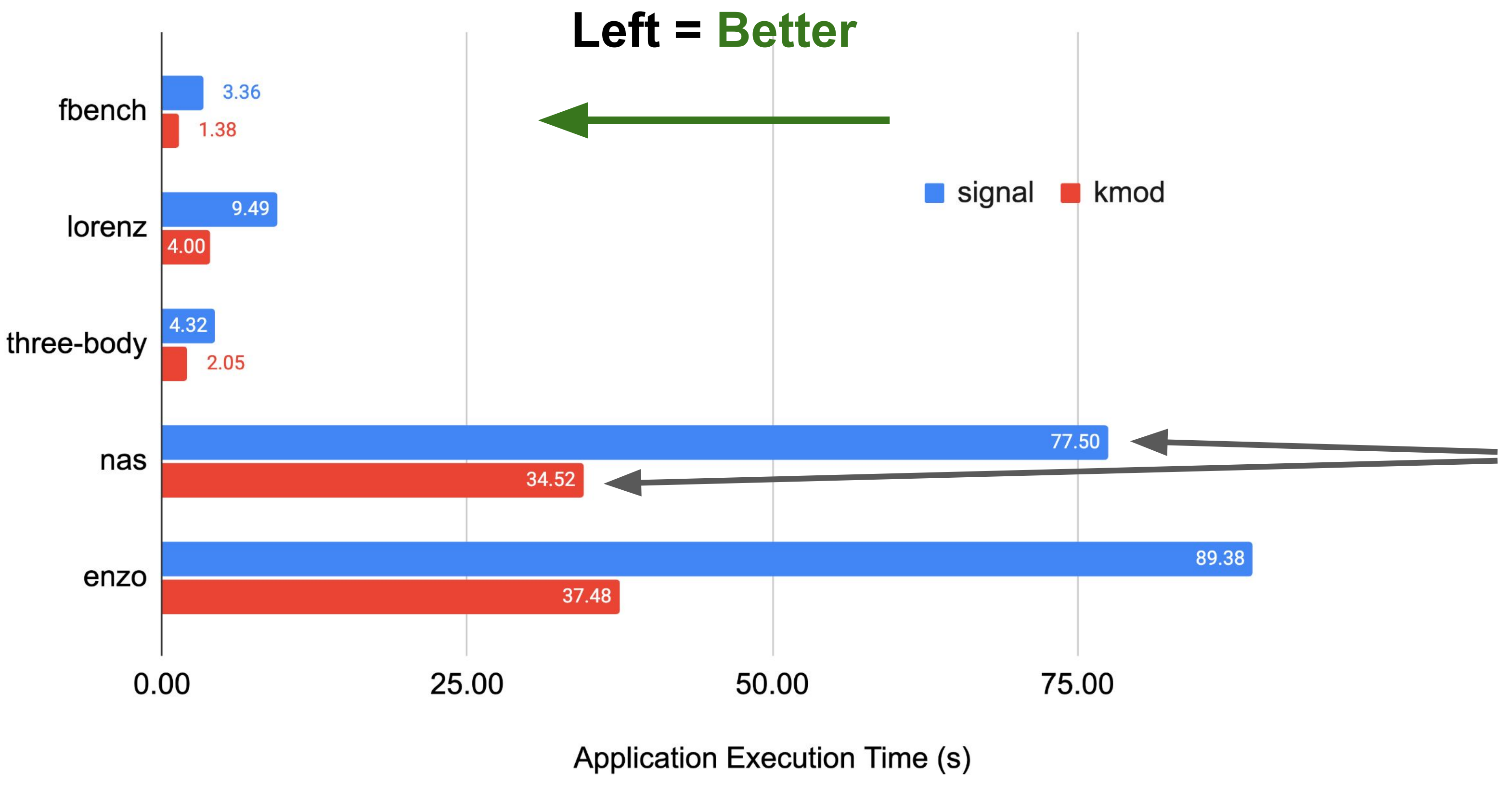
What if you could extend *Linux* with these extensions?

A custom exception handler accelerates FPVM[1]



1. FPVM: Towards a Floating Point Virtual Machine. HPDC 2022

A custom exception handler accelerates FPVM[1]



>50% reduction in runtime when virtualizing floating point in SIGFPE handlers

1. FPVM: Towards a Floating Point Virtual Machine. HPDC 2022

Custom timer delivery accelerates Heartbeat[1] scheduling

A thread scheduler for irregular workloads

A custom kernel module for timer interrupts instead of signals

lowers delivery overhead by 3-4x

(Recently accepted to ASPLOS'24)

1. Mike Rainey, Ryan R. Newton, Kyle Hale, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. PLDI 2021

So Linux kernel modules can accelerate your applications or runtimes!

Why isn’t everyone doing this?

Unfortunately…

Vendors ***really*** don't like you inserting kernel modules

Why?

No hardware protections!

(the module can just turn them off)

Unrestricted access to all of physical memory (+MMIO)

(The module can just rewrite the page tables)

Crashes, Data Corruption

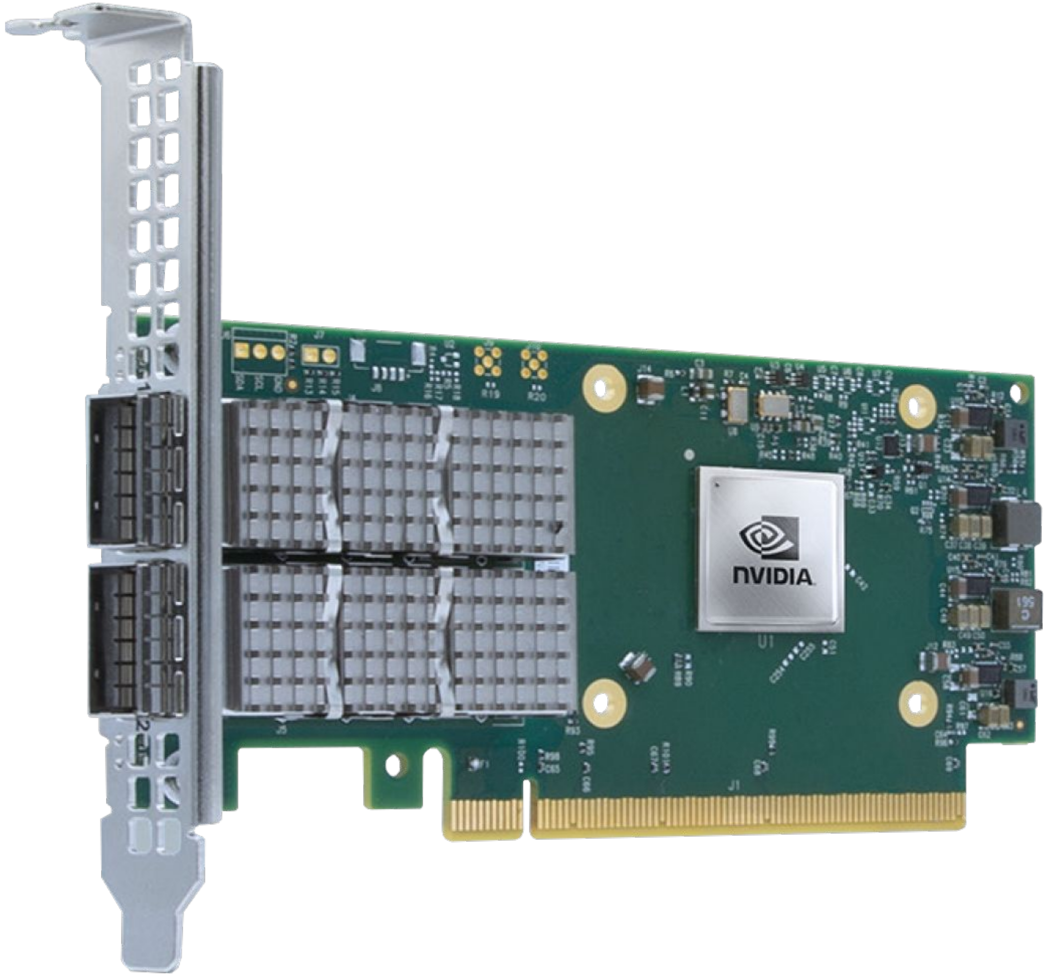
**(A bug in the module can bring the whole kernel down.
No isolation)**

What if Linux could be *safely* *extended* with kernel modules?

Further…

Can Linux be extended with *policies* that dictate what a kernel module can and cannot access?

For example…


NVIDIA


struct ib_cq *cq = container_of(iop, struct ib_cq, iop);
struct dim *dim = cq->dim;
int completed;
completed = __ib_process_cq(cq, budget, cq->wc, IB_POLL_BATCH);
if (completed < budget) {
irq_poll_complete(&cq->iop);
if (ib_req_notify_cq(cq, IB_POLL_FLAGS) > 0) {
trace_cq_reschedule(cq);
irq_poll_sched(&cq->iop);
if (dim)
rdma_dim(dim, completed);
return completed;
static void ib_cq_completion_softirq(struct ib_cq *cq, void *privat
trace_cq_schedule(cq);
irq_poll_sched(&cq->iop);
static void ib_cq_poll_work(struct work_struct *work)
struct ib_cq *cq = container_of(work, struct ib_cq, work);
int completed;
struct list_head id_list;
enum ib_gid_type *default_gid_type;
u8 *default_roce_tos;
struct rdma_bind_list {
enum rdma_ucm_port_space ps;
struct hlist_head owners;
unsigned short port;
static int cma_ps_alloc(struct net *net, enum rdma_ucm_port_space ps,
struct rdma_bind_list *bind_list, int snum)
struct xarray *xa = cma_pernet_xa(net, ps);
return xa_insert(xa, snum, bind_list, GFP_KERNEL);
static struct rdma_bind_list *cma_ps_find(struct net *net,
enum rdma_ucm_port_space ps, int snum)
struct xarray *xa = cma_pernet_xa(net, ps);
return xa_load(xa, snum);
static void cma_ps_remove(struct net *net, enum rdma_ucm_port_space ps,
int snum)
struct xarray *xa = cma_pernet_xa(net, ps);
xa_erase(xa, snum);
enum {
CMA_OPTION_AFONLY,
void cma_dev_get(struct cma_device *cma_dev)
refcount_inc(&cma_dev->refcount);
void cma_dev_put(struct cma_device *cma_dev)
if (refcount_dec_and_test(&cma_dev->refcount))
complete(&cma_dev->comp);
struct cma_device *cma_enum_devices_by_ibdev(cma_device_filter filter,
void *cookie)
struct ib_cq *ib_cq_pool_get(struct ib_
int comp_vector_hint,
enum ib_poll_context poll_ct
static unsigned int default_comp_vect
unsigned int vector, num_comp_vectors
struct ib_cq *cq, *found = NULL;
int ret;
if (poll_ctx > IB_POLL_LAST_POOL_TYPE
WARN_ON_ONCE(poll_ctx > IB_POLL_LAST
return ERR_PTR(-EINVAL);
num_comp_vectors =
min_t(unsigned int, dev->num_comp_v
/* Project the affinty to the device
if (comp_vector_hint < 0) {
comp_vector_hint =
(READ_ONCE(default_comp_vector) +
WRITE_ONCE(default_comp_vector, comp
vector = comp_vector_hint % num_comp_
* Find the least used CQ with correct
* enough free CQ entries
while (!found) {
spin_lock_irq(&dev->cq_pools_lock);
list_for_each_entry(cq, &dev->cq_pools[poll_ctx],
pool_entry) {
* Check to see if we have found a CQ with the
* correct completion vector
if (vector != cq->comp_vector)
continue;
if (cq->cqe_used + nr_cqe > cq->cqe)
continue;
found = cq;
break;

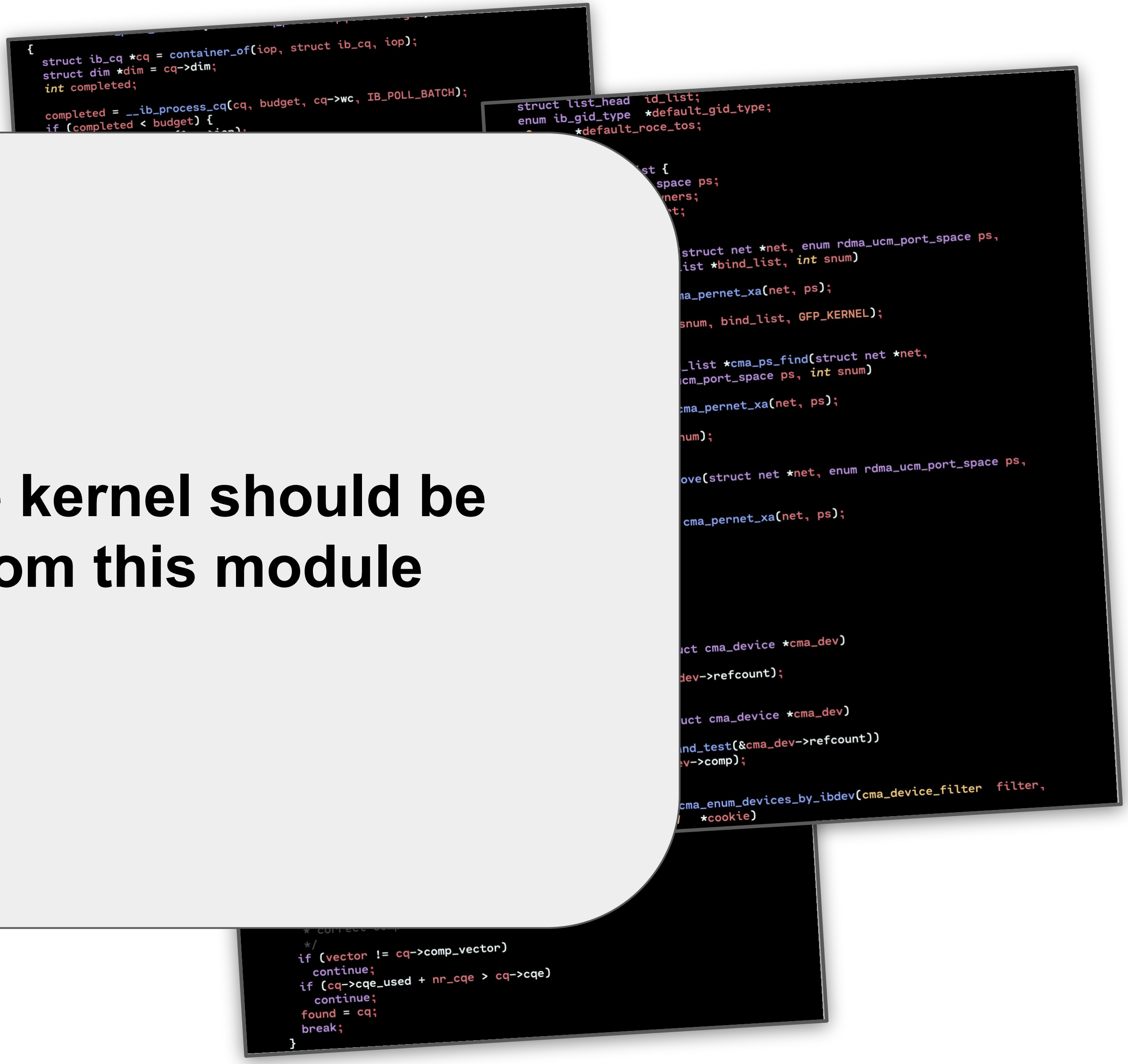
Here’s the trouble

How can the sysadmin be sure it is safe?

Who is going to audit ***all* the code** in the module?

What if the module has **a hidden bug** that you can’t easily see?

For example…



The rest of the kernel should be protected from this module

For example…

A network driver should not be able to change my user id to root!

Such a protection mechanism needs…

To be Fully Automatic

Arbitrary Granularity Memory protection

Extendable with Software Policies

CARAT KOP

Compiler **A**nd **R**untime **A**ddress **T**ranslation

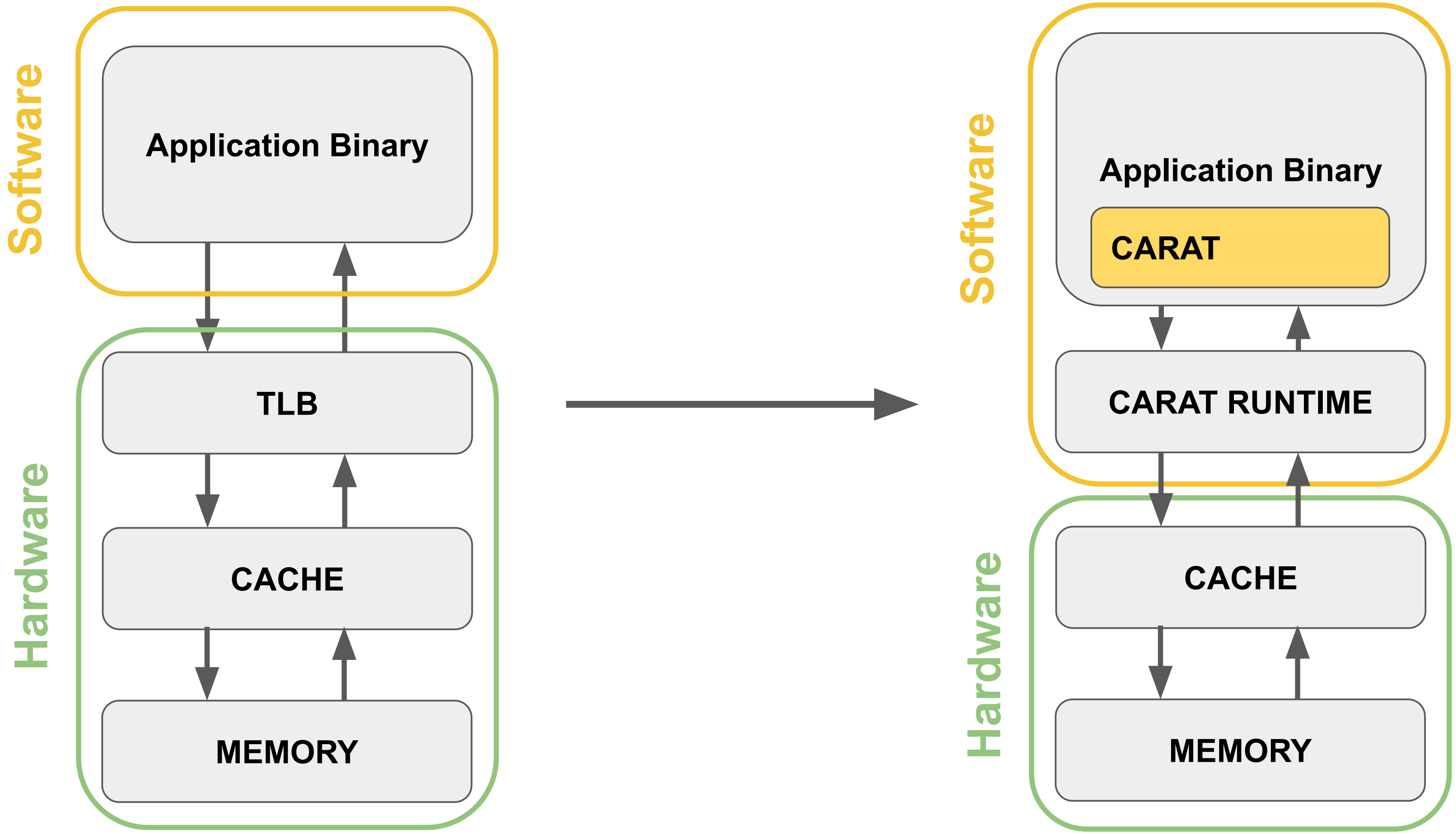
Kernel **O**bject **P**rotection

CARAT KOP

Compiler And Runtime Address Translation

Kernel Object Protection

CARAT: Compiler- And Runtime-based Address Translation[1,2]



1. Suchy et. al. CARAT: a case for virtual memory through compiler- and runtime-based address translation. PLDI 2022
2. Suchy et. al. CARAT CAKE: replacing paging via compiler/kernel cooperation. ASPLOS '22

CARAT replaces paging with a series of capabilities

Memory Protections

Allocation movement

Access pattern tracking

CARAT replaces paging with a series of capabilities

Memory Protections

We focus on just this

Allocation movement

Access pattern tracking

CARAT inserts guards around untrusted memory access

```
struct ib_cq *cq;
int ret = -ENOMEM;

cq = rdma_zalloc_drv_obj(dev, ib_cq);
if (!cq)
  return ERR_PTR(ret);

cq->device = dev;
cq->cq_context = private;
cq->poll_ctx = poll_ctx;
atomic_set(&cq->usecnt, 0);
cq->comp_vector = comp_vector;
```

```
struct ib_cq *cq;
int ret = -ENOMEM;

cq = rdma_zalloc_drv_obj(dev, ib_cq);
if (!cq)
  return ERR_PTR(ret);
carat_guard(cq);
cq->device = dev;
cq->cq_context = private;
cq->poll_ctx = poll_ctx;
atomic_set(&cq->usecnt, 0);
cq->comp_vector = comp_vector;
```

* big simplification

We experimented with a full-system designed around CARAT

In this work, we investigate applying CARAT to *linux*

CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation

Brian Suchy, Northwestern University, Evanston, IL, USA, brian@briansuchy.com
Souradip Ghosh, Northwestern University, Evanston, IL, USA
Drew Kersnar, Northwestern University, Evanston, IL, USA
Siyuan Chai, Northwestern University, Evanston, IL, USA
Zhen Huang, Northwestern University, Evanston, IL, USA
Aaron Nelson, Northwestern University, Evanston, IL, USA
Michael Cuevas, Northwestern University, Evanston, IL, USA
Alex Bernat, Northwestern University, Evanston, IL, USA
Gaurav Chaudhary, Northwestern University, Evanston, IL, USA
Nikos Hardavellas, Northwestern University, Evanston, IL, USA, nikos@northwestern.edu
Simone Campanoni, Northwestern University, Evanston, IL, USA, simonec@eecs.northwestern.edu
Peter Dinda, Northwestern University, Evanston, IL, USA, pdinda@northwestern.edu

ABSTRACT

Virtual memory, specifically paging, is undergoing significant innovation due to being challenged by new demands from modern workloads. Recent work has demonstrated an alternative software-only design that can result in simplified hardware requirements, even supporting purely physical addressing. While we have made the case for this Compiler- And Runtime-based Address Translation (CARAT) concept, its evaluation was based on a user-level prototype. We now report on incorporating CARAT into a kernel, forming Compiler- And Runtime-based Address Translation for CollAborative Kernel Environments (CARAT CAKE). In our implementation, a Linux-compatible x64 process abstraction can be based either on CARAT CAKE, or on a sophisticated paging implementation. Implementing CARAT CAKE involves kernel changes and compiler optimizations/transformations that must work on all code in the system, including kernel code. We evaluate CARAT CAKE in comparison with paging and find that CARAT CAKE is able to achieve the functionality of paging (protection, mapping, and movement properties) with minimal overhead. In turn, CARAT CAKE allows significant new benefits for systems including energy savings, larger L1 caches, and arbitrary granularity memory management.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507771

CCS CONCEPTS

• **Software and its engineering** → **Operating systems, compilers**; **Runtime environments**; • **Blended systems**;

KEYWORDS

virtual memory, memory management, kernel, runtime

ACM Reference Format:
Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. 2022. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3503222.3507771

1 INTRODUCTION

Virtual memory, specifically address translation implemented with paging, is deeply embedded in today's systems at all levels, but particularly within the hardware and the kernel. As we have known since the 1960s [50], virtual memory solves numerous problems. This includes providing a simplifying memory abstraction for programmers, protecting the kernel from processes and processes from each other, and extending physical address space via swapping to/from storage. Its most popular form, paging, also provides a natural unit for memory management.

Unfortunately, paging[1] is not without cost. Paging *requires* hardware/software codesign spanning the hardware directly on the access path to main memory and the deepest levels of the kernel. The hardware structures supporting the traditional address translation model (per-core DTLBs, ITLBs, STLBs, separate structures for different page sizes, nested TLBs, quad pagewalkers, walker

[1] And its kissing cousin, segmentation.

1. Suchy et. al. CARAT CAKE: replacing paging via compiler/kernel cooperation. ASPLOS '22

CARAT KOP

Compiler And Runtime Address Translation

Kernel Object Protection

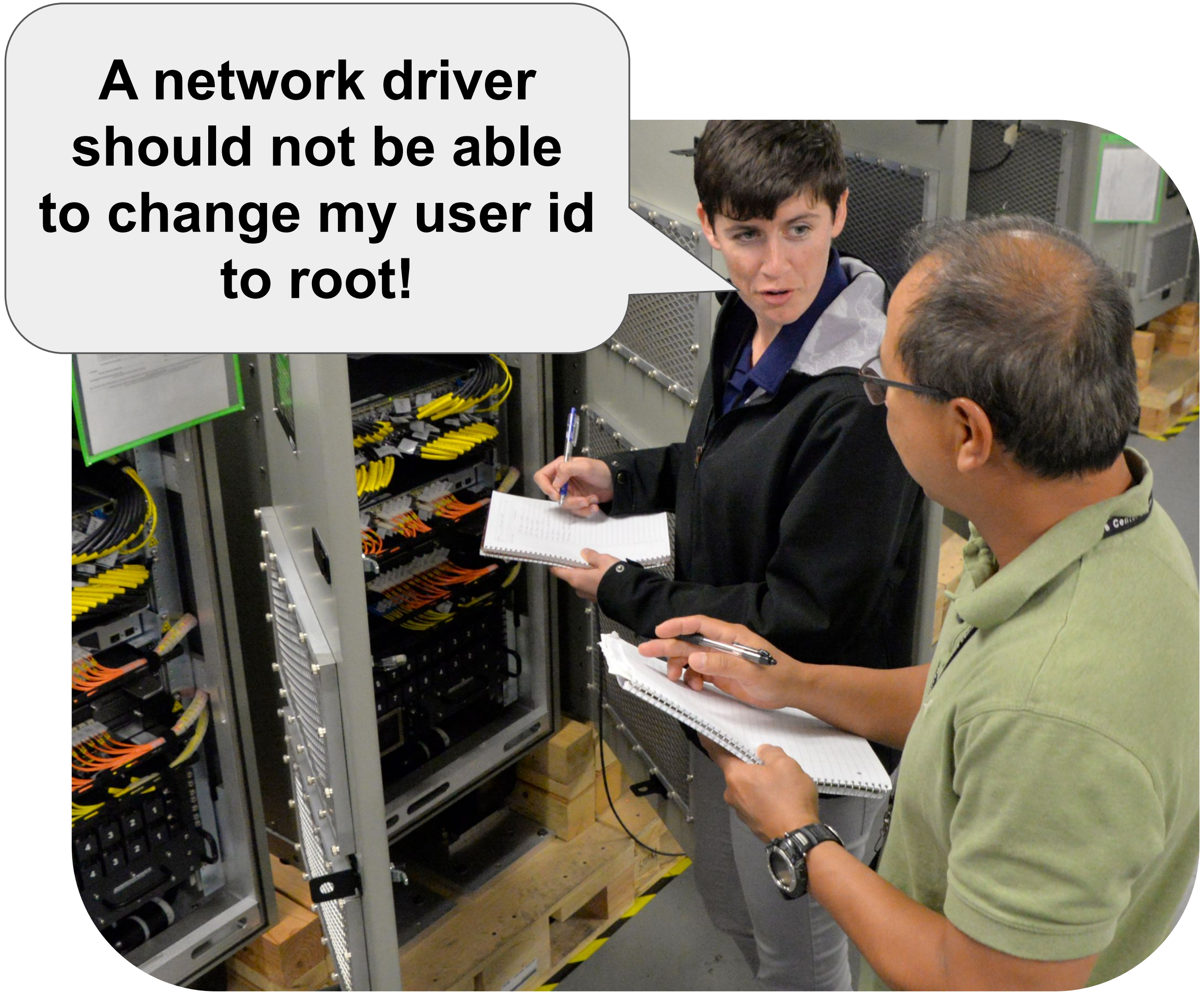
Kernel Object Protection

Allow a policy to dictate memory access - independent of hardware protections

Require **that untrusted kernel modules are compiled with CARAT, ensuring it adheres to this policy.**

We aren’t sure what this policy should look like yet.

Maybe a policy looks like this?

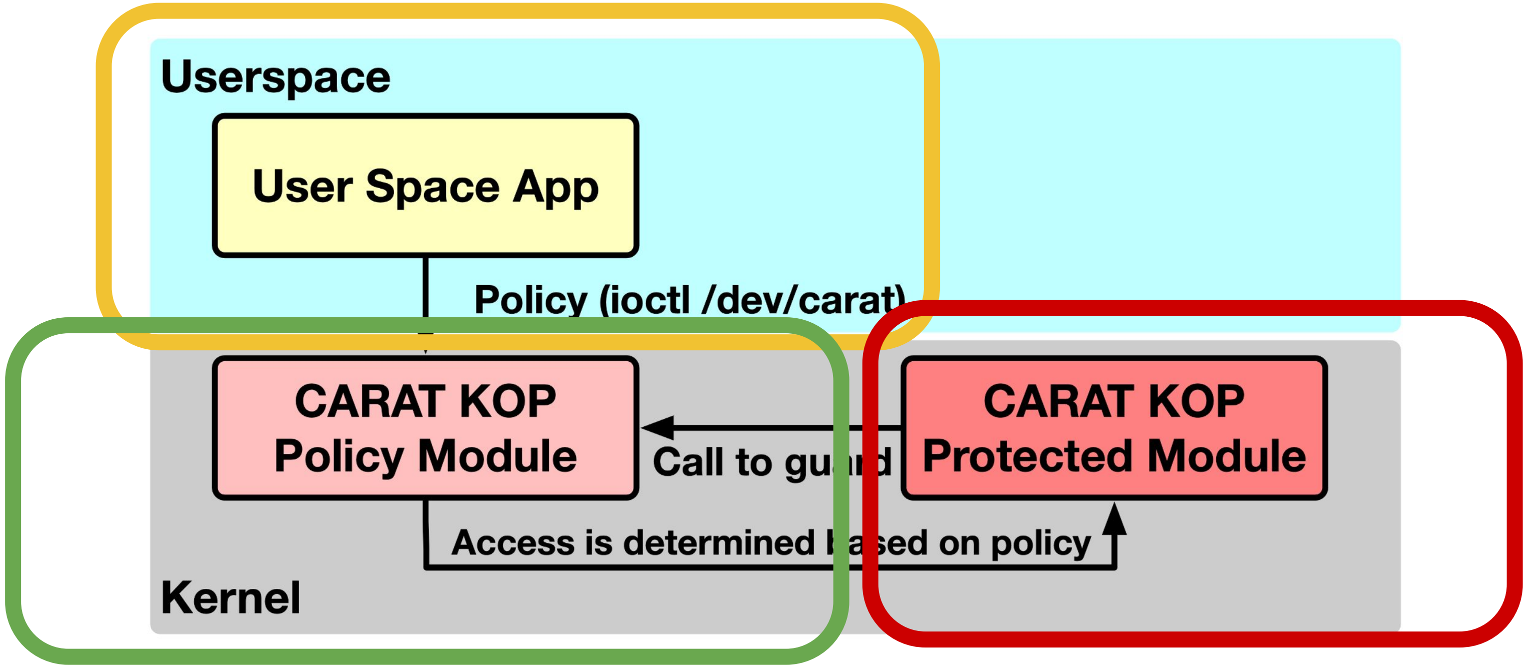


This notion of guarding memory access at any granularity is enough to run *any* memory policy.

Our CARAT KOP Prototype is broken into three parts

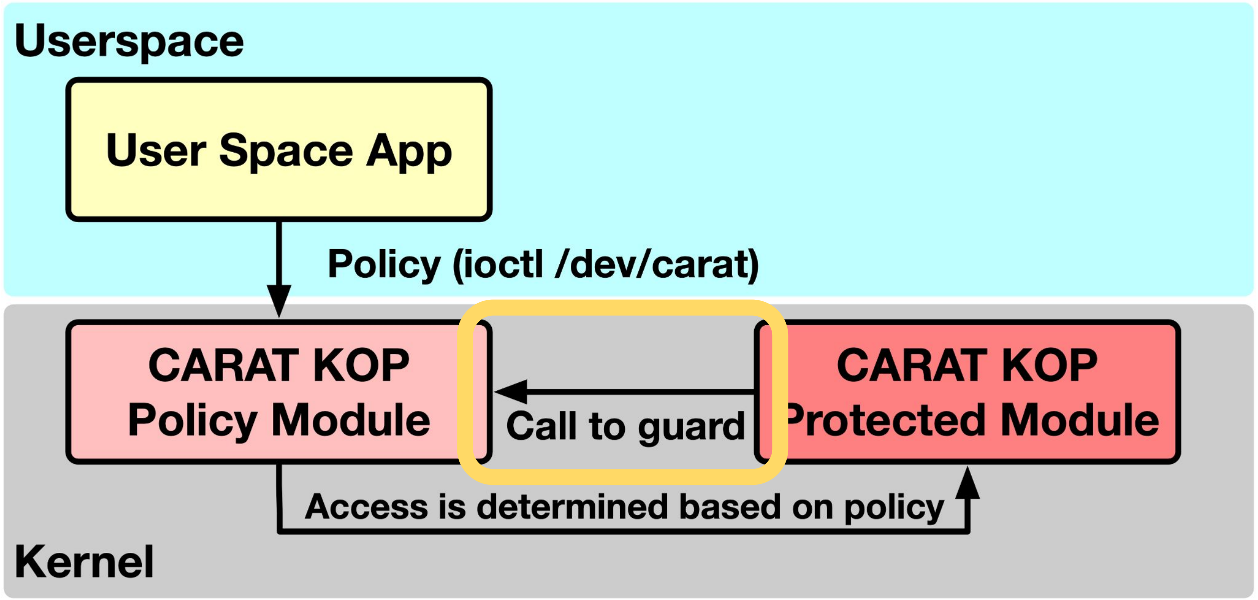
A userspace program to configure the policy.

A module that enforces the policy



And an “untrusted” module which the policy is enforced upon

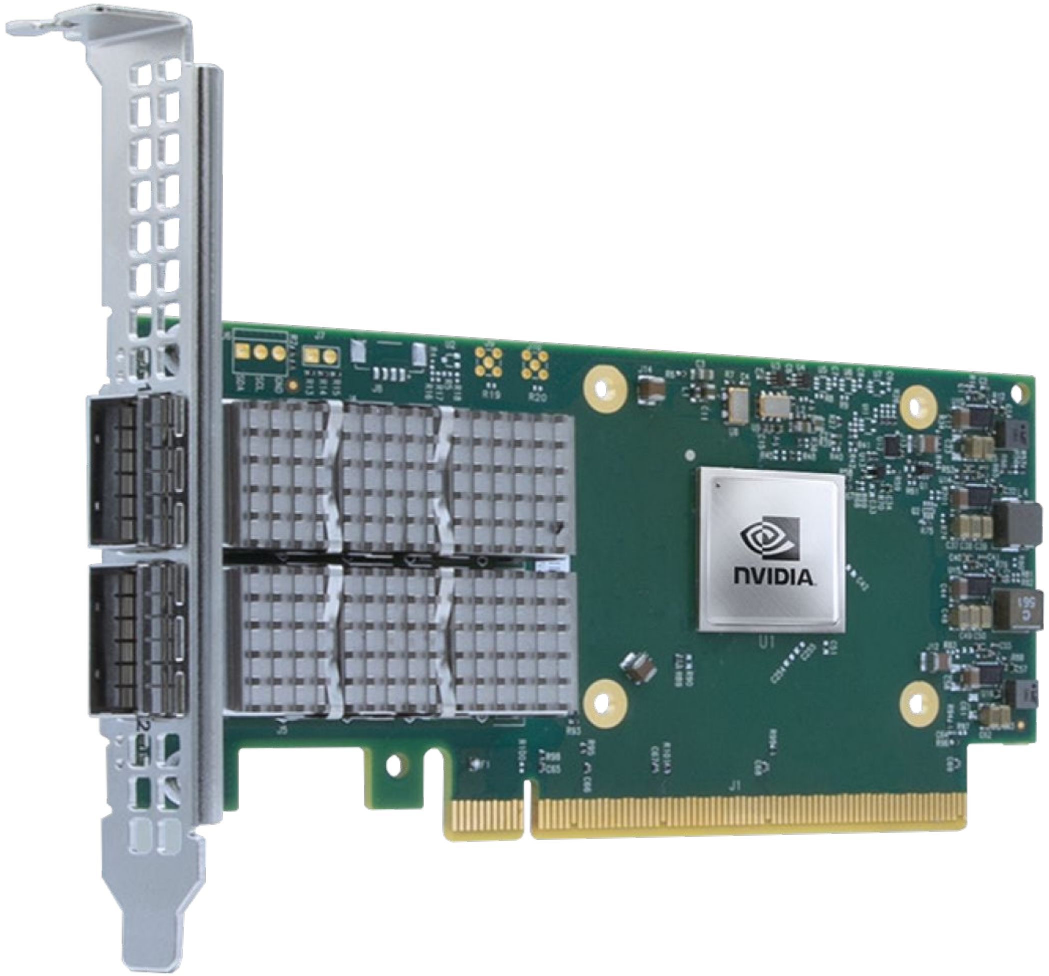
The policy is enforced by calls to `carat_guard()`



```
struct ib_cq *cq;
int ret = -ENOMEM;

cq = rdma_zalloc_drv_obj(dev, ib_cq);
if (!cq)
  return ERR_PTR(ret);
carat_guard(cq);
cq->device = dev;
cq->cq_context = private;
cq->poll_ctx = poll_ctx;
atomic_set(&cq->usecnt, 0);
cq->comp_vector = comp_vector;
```

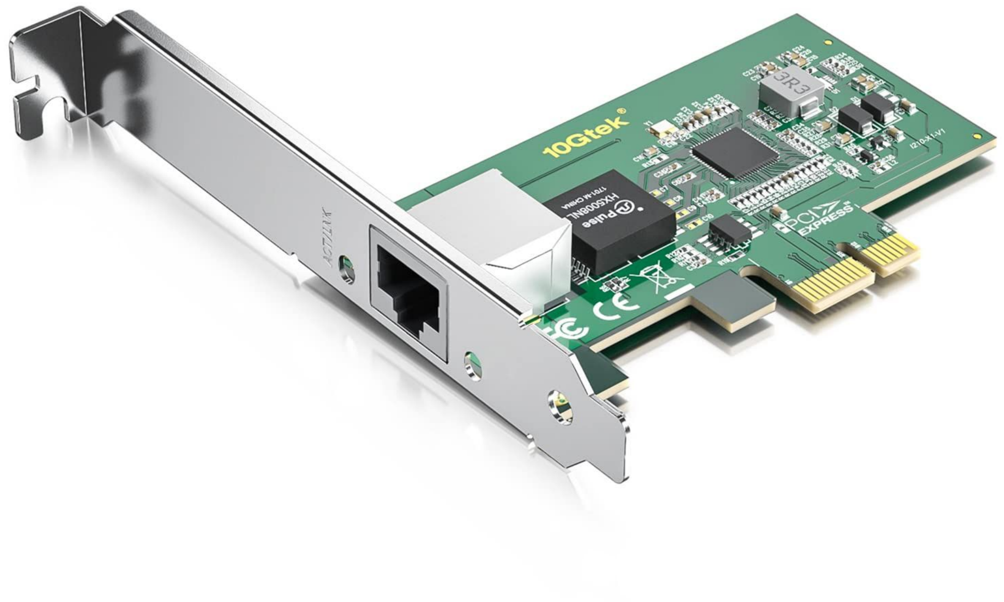
```c
memory_region_t *restricted_regions;
void carat_guard(void *ptr) {
  for (long i = 0; i < region_count; i++) {
    if (contains(restricted_regions[i], ptr) {
        printk("No permission to access!\n");
        abort();
    }
  }
}
```

So what’s the performance like?



Unfortunately, we don’t have access to one of these right now…

Instead…



**As a *proof of concept*,
we instead applied
CARAT KOP to an
unmodified e1000e driver**

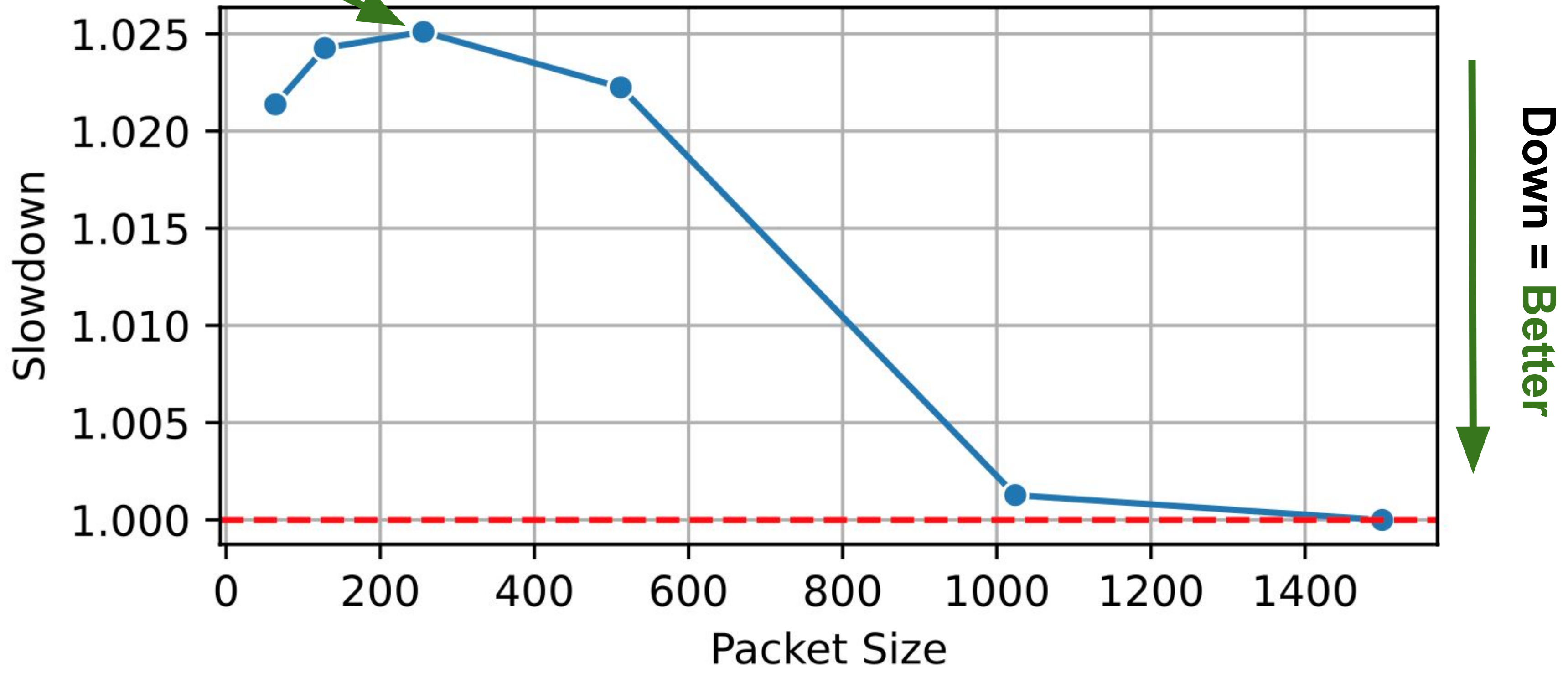
It’s not a trivial driver - around 20k loc

Compiling the kernel module is easy

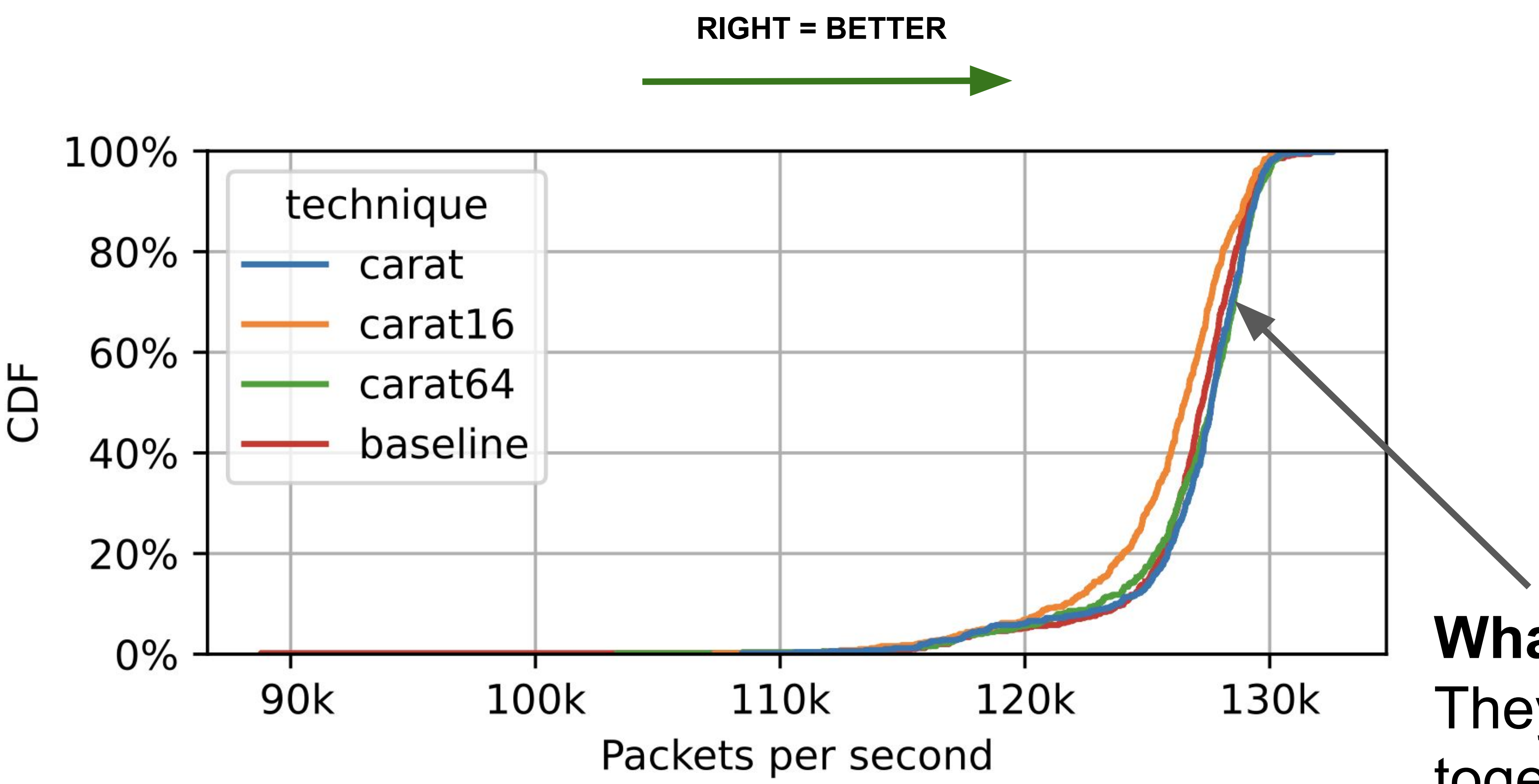
```
obj-m += e1000e.o
ccflags-y := -Xclang -fpass-plugin=$(PWD)/pass/build/CaratKop.so
e1000e-objs := 82571.o ich8lan.o 80003es2lan.o \
         mac.o manage.o nvm.o phy.o \
         param.o ethtool.o netdev.o ptp.o

all:
  make -C $(LINUX) M=$(PWD) modules
```

2.5%

slowdown in the ***worst case!***



We sweeped the complexity of the policy to determine its effect



What's important: They are all bundled together

Conclusion

CARAT KOP **allows Linux to enforce a software policy to the memory accesses in an unmodified kernel module**

With low overhead (~2.5%)

Outstanding Questions

- What should these policies look like?
  - CARAT KOP allows ***any*** policy, it’s just code
- What should happen when a module violates a policy?
  - Currently, we halt the kernel
- Should different modules have different policies?
- How do we validate that a module was correctly compiled with CARAT?
  - We built code-signing for CARAT CAKE
- Does CARAT KOP extend to other modules?

Thank you!

Contact: ncw@u.northwestern.edu

Thomas Filipiuk, **Nick Wanninger,** Nadharm Dhiantravan, Carson H Surmeier, Alex Bernat, Peter Dinda