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

Thomas Filipiuk
Northwestern University
United States
tfilipiuk@u.northwestern.edu

Nick Wanninger
Northwestern University
United States
ncw@u.northwestern.edu

Nadharm Dhiantravan
Northwestern University
United States
nadharm@u.northwestern.edu

Carson H Surmeier
Northwestern University
United States
chs@u.northwestern.edu

Alex Bernat
Harvard University
United States
alexbernat@college.harvard.edu

Peter Dinda
Northwestern University
United States
pdinda@northwestern.edu

ABSTRACT

Extending Linux through the kernel module interface can offer immense benefits and capabilities in high performance computing (HPC). These extensions can also be more readily deployed because Linux is the common, typically only, supported OS choice among supercomputing vendors. However, because Linux is monolithic, Linux kernel modules are free to read and write any address with kernel-level permissions. A poorly written—or untrustworthy—module can wreak havoc on the whole system. This unfortunately means that many production HPC systems often do not permit custom kernel modules to be inserted into the system, no matter the benefit.

By limiting what objects in the physical address space the module can have access to, it may be possible to guarantee memory safety for these modules. In this paper, we discuss the possibility of using the previously developed compiler- and runtime-based address translation (CARAT) model and toolchain to inject guards around a kernel module's memory accesses. The memory accesses would then be allowed or disallowed according to a memory access policy specified by the user, in what amount to firewall rules. We share our results regarding the guard injection and address validation process. The CARAT-based Kernel Object Protection (CARAT KOP) prototype is able to transform a substantial production kernel module from the kernel tree (a network device driver composed of approximately 19,000 lines of code). The transformed module can then run with minimal effect on performance while restricted to a set of address regions.

CCS CONCEPTS

• **Software and its engineering → Operating systems, compilers**; **Runtime environments**; • **Security and privacy → Systems security**; • **Blended systems**;

KEYWORDS

protection, kernel module, kernel, Linux

ACM Reference Format:
Thomas Filipiuk, Nick Wanninger, Nadharm Dhiantravan, Carson H Surmeier, Alex Bernat, and Peter Dinda. 2023. CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624237

1 INTRODUCTION

High performance computing environments can benefit from the performance or capabilities afforded by specialized kernel support. Although custom kernels [9, 14, 16, 18, 28, 31], perhaps combined with hardware partitioning support [13, 27], are a powerful way to do this, in many cases it is possible to add the capability as kernel module for Linux [11, 17]. We have ourselves developed Linux kernel modules for fast high-performance floating point trap delivery as part of FPVM [6], and fast timer delivery for heartbeat scheduling [29].

Beyond being able to leverage Linux to avoid the major engineering involved with specialized kernels, the Linux kernel module route also bodes well for deployment. It has proven difficult to get supercomputing vendors to supply, and thus make easier to deploy, custom kernels. It is much more straightforward for them to deliver what are ultimately

ROSS '23, November 12, 2023, Denver, CO

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3624062.3624237

variants of Linux. Deploying Linux kernel modules on top of these is a much smaller ask. Indeed, the vendor might even supply the kernel build framework, much like in commodity Linux, and thus allow the construction of custom modules with the choice of inserting the modules being that of the supercomputer's operators.

Why might a supercomputer operator be wary of deploying custom Linux kernel modules? Because they are inherently dangerous. The Linux kernel is monolithic, and consequently kernel module code operates with no real restrictions. The boundary between the core kernel and the module is essentially nonexistent. While there are several ways [12, 32] in which a module could subvert the kernel (or simply have a bug that causes a problem), in this work we focus on one: memory access.[1]

A kernel module can generally access any part of memory, including regions critical to the operating system. This means that an action as simple as installing a custom device driver module can come with very high risks if not done securely. The consequences of installing buggy or malicious modules into the kernel can range from corruption of data to full-fledged rootkit-style attacks. Thus, Linux users need to take care to only insert module code from trusted vendors or otherwise verify the security of modules before installing them. This makes difficult the goal of being able to incorporate custom modules to benefit HPC.

One way to potentially mitigate the security threat of kernel modules would be to limit the addresses they may use without revoking their kernel-level privileges. This is our focus here. We present evidence that this is feasible with compiler transformation combined with special runtime support implemented as a kernel module. The core kernel is left untouched, while the kernel module must be recompiled but requires absolutely no source changes.

Our prototype CARAT KOP (CARAT-based Kernel Object Protection) system leverages technology developed for the compiler- and runtime-based address translation (CARAT [33, 34]) project. CARAT and CARAT CAKE seek to replace the hardware/kernel co-design of traditional paging-based virtual memory, with a compiler/kernel co-design that avoids paging. CARAT KOP uses a variant of the guarding process from CARAT, as applied to Linux kernel modules, and operationalized within the Linux kernel. Essentially, we inject callbacks to a guard function into the kernel module code using a clang compiler (version 14.0.0) pass. This guarding process is certified by the compiler, and can be validated by the kernel when the transformed module is inserted. Then, during the runtime of the module, the guard function checks memory accesses against a policy provided by the user. If the permissions required of the access and the permissions provided in the policy do not align, the access is disallowed.

CARAT KOP's guard injection method can be used to create protective barriers between Linux kernel modules and the rest of the kernel, using default allow or default deny policies. This would provide a guarantee of safety (at least to some degree) when installing custom modules, which Linux currently lacks. The guarantee would be with respect to policies set by the operator.

Our contributions are:

- We argue for the need for core kernel protection from custom kernel modules in order to enable the deployment of specialized modules in HPC environments.
- We describe the design and implementation of CARAT KOP, a proof-of-concept system for achieving such protection through compiler transformation and specialized runtime support.
- We apply CARAT KOP to a substantial Linux kernel module, a network driver, to demonstrate its capability.
- We evaluate the performance of the transformed module and find it is only minimally effected.
- We lay out the many unresolved questions about the exact mechanism of the memory guards, the creation of memory region policies that are both practical and secure, and trade-offs in terms of performance.

2 CARAT AND CARAT CAKE

We leverage technology developed for the compiler- and runtime-based address translation, or CARAT project. The original CARAT paper [33] proposed a software-only alternative to the traditional hardware-based virtual memory abstraction that uses compiler and operating system co-design instead of hardware and operating system co-design. In effect all code in the system is transformed by the compiler (we use LLVM) to introduce additional code that then interacts with a special runtime component or directly with the kernel. This combination provides the capabilities of protection/isolation and memory object movement at arbitrary granularity without using paging. This is accomplished by three basic mechanisms: (1) allocation and pointer escape tracking, (2) pointer patching, and (3) guards. (1) and (2) are used to support memory object movement and thus are irrelevant to CARAT KOP.

CARAT CAKE [34], Compiler- And Runtime-based Address Translation for CollAborative Kernel Environments, is an implementation of CARAT that combines a highly optimized version of the compiler transformations (based on NOELLE [20]) and specialized support within the Nautilus

[1] By memory access, we mean access to the physical address space, and RAM, MMIO devices, etc, that are mapped into it. Note that on Linux, virtual memory is always in effect in the core kernel, but the physical address space is remapped in the kernel to be accessible at a known offset in the virtual address space.

kernel [14] to *safely run Linux user executables within the kernel.* The CARAT CAKE mechanisms are leveraged to create a Linux-compatible process abstraction, and to protect the kernel from the process and processes from each other, all without any paging-level protections. Note that this model has a clear analogy with Linux kernel modules, where the Linux kernel modules are akin to CARAT CAKE processes. It is the protection capability of CARAT CAKE, through guards, (3) from above, that we employ in CARAT KOP.

CARAT CAKE accomplishes memory protection for its processes through the compiler's insertion of guards before each load or store in executables.[2] Guards are simply callbacks to a CARAT CAKE runtime function that is privately exported from the kernel. This function performs a permissions check and prevents unauthorized memory accesses, squashing them into something similar to a page fault. It is important to understand that CARAT CAKE guards operate at arbitrary granularity, meaning that protection is possible down to individual bytes. CARAT KOP inherits this benefit.

Of course, guards are a very expensive proposition if, indeed, we checked each data reference. In fact, in CARAT CAKE, it is only through extensive compiler analysis that we are able to generally hoist such guards and amortize them across many references, even in challenging application benchmarks like SPEC, PARSEC, NAS, and others. The guard function itself is also highly optimized for the common case, inlined everywhere, and then whole program optimization is applied, producing a statically linked executable.

Guards in CARAT KOP can be much simpler. As CARAT KOP focuses on kernel module protection rather than user code, we do not expect that the significant compiler analysis and optimization of guards will be necessary to achieve low overhead, as kernel modules tend to implement less complex algorithms. Further, because we are not attempting to replace paging in CARAT KOP, we can fall back on the Linux kernel's use of traditional hardware-based virtual memory for some enforcement. For example, paging can be used to mark the kernel module's code pages as unwritable, thus avoiding the problem of self-modifying code.

In CARAT CAKE, the compilation processes also performs cryptographic code signing. This is then used at load time to prove to the kernel that the proper processing has been performed (e.g., that guards have been injected) and by which compiler. The signature also is in effect an assertion, by the compilation process, that the code it compiled does not include any problematic elements such as inline or separate assembly. These limitations are discussed further in §5. Undefined behavior at the C/C++ level is a non-problem because all transformations operate at the LLVM middle-end, at which point any front-end language behavior has been lowered out, and LLVM IR itself does not include undefined behavior. CARAT KOP needs a similar code signing and validation process.

[2]CARAT CAKE provides control flow integrity (e.g., instruction fetch guarding) through a different mechanism that is irrelevant to CARAT KOP, which does not attempt to guard instruction fetches.

3 CARAT KOP

We now break down the implementation of the various parts that make up the CARAT KOP system, and how they all fit together. The system comprises three logical components: the kernel module which enforces a memory access policy (§3.1), the protected kernel modules which must abide by said policy (§3.2), and the compiler which transparently inserts runtime checks into those modules (§3.3).

3.1 Policy Module

Perhaps the most important component of CARAT KOP is the *policy* module. Simply, this module is inserted into the kernel and provides a single symbol, `carat_guard`, which is invoked by modules which have been transformed by the compiler. This interface is general enough—and simple enough—that potentially any memory policy system could be built on top of it. The signature of the guard function is as follows:

```
void carat_guard(void*  addr,
                 size_t size,
                 int    access_flags);
```

Before a memory access performed, this function is invoked with the address, access size, and a bitmap of flags that indicate the intent of the access (read/write). This information can be compared against whatever data structure or restrictions that the policy module might require.

In this work, we evaluate against a relatively simple policy module which is based on a table of *regions*. As seen in Figure 1, a root user can communicate with the policy module through an `ioctl` system call to add or remove regions from the table using a simple application, `policy-manager`.

We use a table describing a maximum of 64 memory regions and thus a permissions check has $O(n)$ time complexity. A table was chosen in order to minimize pointer chasing, lending speedup over other implementations like the Linux kernel's red-black tree (even though the tree would have $O(\log n)$ time complexity). Each entry stores a region's lower bound, length, and protection flags. When the guard function is invoked, the policy module then simply walks the region table and checks if the access should be permitted.

Whilst the table is effective with a reasonable amount of performance overhead, there is significant potential to improve this policy structure. Probabilisitic structures, like any

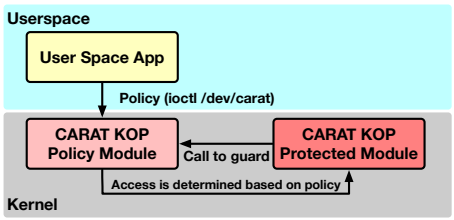

Figure 1: A server owner can configure the CARAT KOP policy through the ioctl interface.

of a variety of AMQ-filters [2, 8, 36], may very well improve average performance, as we expect modules to be compliant with policies for nearly every access, significantly reducing the number of policy table lookups needed. Modification of the table to use a locality-sensitive hash function, thus finding the "closest bucket" of policy-defined regions to an arbitrary address in constant time, would also be a potential optimization. The primary tradeoff in modification of the policy data structure is the inability to maintain overlapped regions. Importantly, CARAT KOP *does not* attempt to define an optimal policy or method of policy checking, but provides the methodology to easily iterate upon a simplistic structure, the 64-entry table.

While this *specific* policy implementation is incredibly simple, it provides enough capability such that a kernel module could be restricted from accessing regions it should not be permitted. For example, the module could be configured to block access to the direct-mapped physical memory with a single rule. Or, it could restrict access to the heap to be read-only.

When the policy module determines that a memory access should be forbidden, the expected behavior would be to halt execution of the module and unload it before damage can be done. Unfortunately, the kernel module interface is extremely complicated. Forcefully killing and ejecting a malfunctioning module while it is running can be incredibly dangerous — especially if the module uses concurrency control mechanisms like mutex locks. For example, if the troublesome module takes a global lock, then gets unloaded, the system may eventually deadlock as the lock was not released. This problem *could* potentially be solved by utilizing higher level concepts such as exceptions. When paired with the "RAII" programming model of that language, exceptions thrown by the `carat_guard` function can offer a clean way to free resources when exceptions are raised.

Unfortunately, most of the kernel is written in C and does not feature any exception systems. As such, in this work we currently do not cleanly handle forbidden accesses, and instead log that they occur and cause a kernel panic.

It is important to point out that a kernel panic is actually a reasonable response for the HPC use cases we focus on here (§1). If a specialized kernel module is being brought into a production environment, and a guard check fails, then there are one of three possibilities: (1) the policy set by the operator is incorrect for the module, (2) the kernel module has a bug, or (3) the kernel module is attempting to mount an attack. In all three situations, a hard stop *should* happen.

3.2 Protected Modules

The second part of CARAT KOP is the protected modules themselves. These modules undergo the compiler transformation outlined in §3.3, and as a result have a memory access policy enforced on them because they call back into the policy module. Any module in the Linux kernel can be compiled as a protected module by swapping the compiler for the CARAT KOP compiler. Out-of-tree modules, which are the majority of expected HPC modules, can also be readily compiled as protected modules using the standard out-of-tree processes and a change of compiler.

When a protected module is inserted into the kernel (after validating its signature), it is linked against the policy module's implementation of `carat_guard`. This allows one guard function to be swapped for another without having to recompile the guarded module. Once inserted, the module must adhere to the policy set by the policy module.

3.3 Guard Injection

CARAT KOP compilation is a variant of CARAT CAKE compilation (§2)). Similarly, the CARAT KOP compiler acts on top of the LLVM compiler infrastructure, and operates on the middle-end. The compiler was built on top of an unmodified clang 14.0.0. It is important to note that by "compiler", we essentially mean a compiler pass that lives within the LLVM framework. It is separately compiled from the core compiler, and invoked by a script that wraps the underlying clang compiler. Consequently, our compilation process could be applied within any modern LLVM framework that is sufficient to compile the Linux kernel version we are considering (and hence modules for it). We chose clang/LLVM 14.0.0 because of the particular kernel version we target for testing (5.17.5).

To enable proper guarding of memory accesses, CARAT KOP utilizes a compiler transformation that ensures that guard calls are inserted before loads and stores. To ensure guards are inserted, it simply iterates over each load/store operation and insertes a call to the guard function before. Unlike CARAT CAKE, CARAT KOP does not currently optimize guards—every memory access results in a guard, even if it would be redundant. As we show later (§4), despite the resulting overabundance of unoptimized guards, the performance impact is minor.

We do not optimize guards for engineering reasons. CARAT CAKE guard optimizations leverage the NOELLE analysis

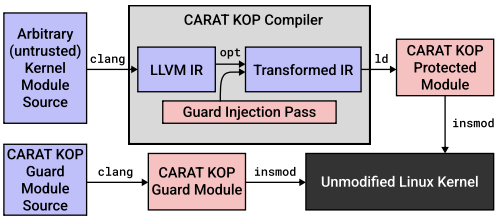

Figure 2: Compilation process.

infrastructure [20], which requires a self-contained "whole program". Kernel modules are, by design, not "whole programs", but rather are effectively a limited kind of shared objects intended to be linked into the kernel. On the positive side, the elimination of analysis and optimization means the resulting CARAT KOP transforms constitute only about 200 lines of C++.

4 EXAMPLE: E1000E NIC DRIVER

As a proof of concept, and to evaluate overheads, we selected a network interface driver from the core kernel tree, and built it as a module, both with and without CARAT KOP. We then test the throughput and latency of packet launches, considering both packet size and number of regions involved in the CARAT KOP policy.

The specific driver we chose was for the "e1000e" family of Intel 1 Gbit/s Ethernet cards. We choose this driver because it is not insubstantial and is part of the mainline kernel already and thus well bug-fixed.

While it is the case that HPC environments typically use much faster network devices, our purpose here is to quantify the difficulty in applying CARAT KOP to existing kernel modules, and to quantify its runtime impact on performance. Recall that CARAT KOP guards memory references. In the e1000e, as with much faster NICs, the overwhelming amount of data transfer occurs due to the DMA engine on the NIC, which is not checked (and thus not slowed) by CARAT KOP.[3] The number of memory references made by the driver to, for example, construct packet headers and transfer descriptors, queue transfer descriptors, and access MMIO device registers is not substantially different between an e1000e and a much faster NIC.

4.1 Engineering effort

We built a BusyBox/initrd-based "mini-linux" distribution around kernel version 5.17.5. This distribution simply boots to a text-based command prompt with no daemons or services running, thus allowing us to have a controlled environment that we can boot the machine from via a USB stick. We built the distribution using clang instead of gcc, although that is not a requirement of CARAT KOP. In a deployment situation, the existing Linux distribution would be used instead, with no effort.

The e1000e driver in the Linux tree comprises about 19,000 lines of source code. In our proof of concept, we extracted the driver out of the kernel tree and set it up to be compiled as a separate module using standard mechanisms, reconfigured our kernel to not include its own e1000e driver. We built two versions of the driver, one with the CARAT KOP transformation applied, the other without it. In both cases, the same compiler was used, with the same flags. No code was modified in the driver. If we were applying CARAT KOP to a specialized HPC module, that module would already exist separately from the Linux tree, and thus CARAT KOP could be applied with a simple recompilation.

In short, the engineering effort needed to use CARAT KOP for a new kernel module is virtually non-existent.

4.2 Performance

Testbed. We tested the performance impact of CARAT KOP on two machines. The first is an outdated Dell R415 containing dual 2.2GHz AMD 4122 processors (each has 4 cores, 256 KB L1i/L1d, 2 MB L2, 6 MB L3) and 16 GB of DRAM. The second is a current Dell R350 containing a 2.8 GHz Intel Xeon E-2378G processor (8 cores, 16 threads, 256 KB L1i/L1d, 2 MB L2, 16 MB L3) with 32 GB of DRAM. Our test NIC is an Intel CT (EXPI9301CTBLK) PCIe board that contains an Intel 82754L chipset. This is attached to a packet sink.

Methodology and factors. We bring the NIC up on a private IP address, and then test using a user-level tool that sends raw Ethernet packets to a fake destination. The tool can vary the number of packets sent and the size of the packets. The tool measures the throughput of the packet transmissions, and the latency of individual packet launches.

Another factor we can vary is the number of regions that are checked by the CARAT KOP policy module. Recall from §3.1 that we currently do an $O(n)$ search across n regions during a guard check. This simple search model is optimized for cache-friendly search of a small number of regions. If a policy scheme wanted to consider many regions, an $O(log(n))$ model could clearly be employed instead.

Minimal impact on packet throughput. Figure 3 shows the effect of using CARAT KOP on packet send throughput. Here, we use the R415 machine. We are sending 128 byte packets, which we believe will show the most significant effects since the guarded portion of the packet launch is amortized over

[3] The natural way to control memory access from DMA is using a technology like the IOMMU or SR-IOV, and is outside the scope of this paper.

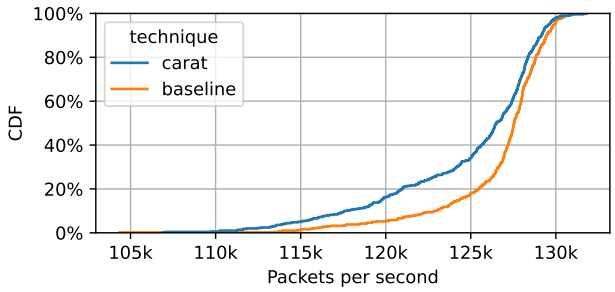

Figure 3: CARAT KOP effect on packet launch throughput on slow R415 machine. Two regions are used. Packet size is 128. The effect is minimal.

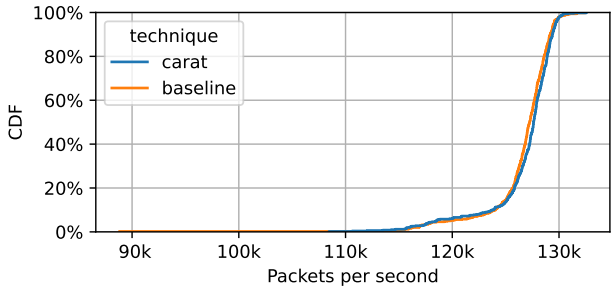

Figure 4: CARAT KOP effect on packet launch throughput on faster R350 machine. Two regions are used. Packet size is 128. The effect is even smaller, and, indeed, almost unmeasurable.

only a small DMA transfer.[4] Our policy is configured to include two memory regions.[5] We run many trials, launching about 100,000 packets per trial. The figure plots the CDF of these trials. "Baseline" refers to the unmodified e1000e module, while "carat" refers to the transformed e1000e module combined with the policy module.

As can be clearly seen, the impact of CARAT KOP is very minimal. The median throughput changes by only about 1,000 packets per second, a relative change of <0.8%.

Figure 4 repeats this experiment on the faster R350 machine. Here, the impact is even smaller. The relative change in the median is <0.1%.

We speculate that the reduced impact on the newer machine is due to a combination of improved caching, branch prediction, and speculation. In the common case, the control flow path for guards introduced by CARAT KOP is incredibly predictable. The guard call will happen, and the region check processing's branches will generally go the same way, and the region check will succeed.

[4]As we will see later (Figure 6) this is a packet size that does indeed amplifies the difference between CARAT KOP and the baseline.

[5]For two regions specifically, the policy rule is that kernel addresses (the "high half") are allowed, but user addresses (the "low half") are disallowed.

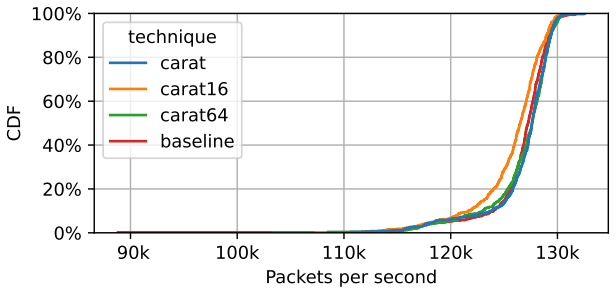

Figure 5: Effect on throughput of varying the number of regions in the CARAT KOP policy on faster R350 machine. Packet size is 128. The effect exists, but is small. For very large numbers of regions a $O(log(n))$ algorithm would ameliorate the effect seen here.

Impact as as a function of the number of regions. In Figure 5, we repeat the previous measurement, but we vary the number of regions, n, contained in the policy. The label "carat64" refers to using CARAT KOP with $n = 64$ regions in the policy. As shown, n does have a small, but significant effect. Even with the worst measured case, however, the relative change to the median is again <1%.

It is important to point out that for all the curves in the figure, the exact same number of guards are being executed. The difference is in the cost of the policy lookup within the guard. We speculate that our cache-friendly linear search is probably effective for policies with 64 regions and perhaps more. At some point, the algorithmic inefficiency will start to dominate, and it would make sense to switch to a logarithmic search data structure. The first of these would be simply to sort the regions in the policy in order, and then do a binary search over the table instead of a linear scan.

The common case is that we will land in a region contained in the policy. It also stands to reason that the regions of a policy will vary in popularity. Consequently, with a large enough number of regions, a popularity-based data structure such as a splay tree or a simple cache over the region data structure (as done in CARAT CAKE) might be able to do better than a logarithmic search in the common case. Other ideas for how to enhance region lookup are given in §3.1.

Impact as as a function of the packet size. In Figure 6 we fix the number of regions to $n = 2$ and vary the packet size (64, 128, 256, 512, 1024, and 1500 byte packets.) For each size, we then determine the average slowdown in the throughput. We see that CARAT KOP's impact is indeed largely independent of the packet size as speculated earlier. To the extent the slowdown varies (maximum is about 2.5%) it is concentrated on small packets.

Impact on packet launch latency. Previously, we have described the effects on packet launch throughput. Figure 7

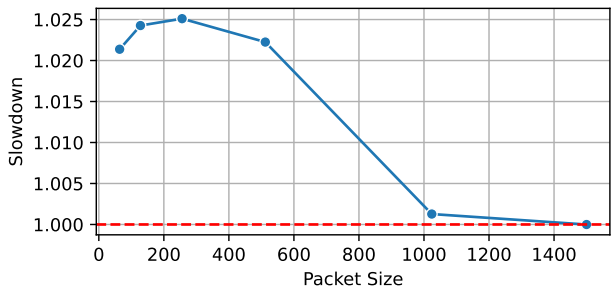

Figure 6: Effect on throughput of varying packet size in the CARAT KOP policy on faster R350 machine.

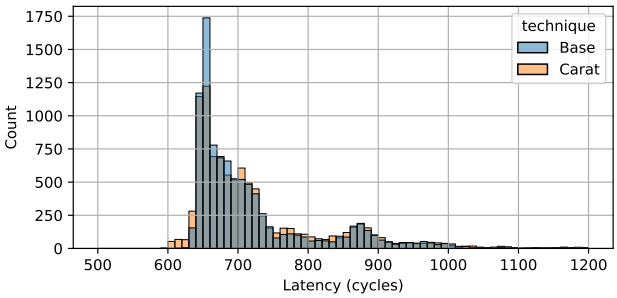

Figure 7: CARAT KOP effect on packet launch latency on faster R350 machine. Two regions are used. Packet size is 128. Outliers (for both CARAT KOP and the baseline) are not included. See text.

shows the effect of CARAT KOP on packet launch latency on the R350 machine. To be specific, we are using a two region policy and 128 byte packets. The latency is measured, in cycles using the cycle counter, as the time spent in the `sendmsg()` call from the user-space test application's point of view. The figure shows the histograms of these times for the CARAT KOP and baseline cases. As can be seen, these are closely matched.

It is important to note that what is being measured here is effectively the cost of a system call and (usually) the time needed to queue a set of transmit DMA descriptors on a ring buffer leading to the 82754L. In the figure, we have hidden the outliers (which can be in excess of 10 million cycles) that occur when the ring is full and the test application is descheduled. Including these outliers, the median times are 694 cycles (CARAT KOP) and 686 cycles (baseline), which is within the measurement noise of the cycle counter.

5 LIMITATIONS AND FUTURE WORK

CARAT KOP's memory guarding mechanism could be extended to restrict kernel module access to files by safeguarding memory regions associated with file system metadata or inodes, as well as a variety of other kernel internals. By delineating and then guarding the memory addresses that contain the mapping and access control details of specific files, CARAT KOP could effectively prevent unauthorized file operations by a kernel module. Similarly, for inter-process communication (IPC), the system could enforce policies by guarding memory regions linked to IPC mechanisms, such as message queues or shared memory segments. If a kernel module attempted an unauthorized IPC action, the CARAT KOP's guard would intercept this by detecting an unauthorized memory access attempt, thereby preventing the communication from taking place. By extending its memory access control techniques to the memory structures integral to file systems and IPC, CARAT KOP could impose granular access controls over a module's file and communication operations. Adding restrictions to additional kernel components could be done incrementally, without specific shared-state algorithms that depend on the topology of specific kernel components. However, these extensions would require a more scalable way to handle many memory regions.

As of now, CARAT KOP does not attempt to prevent access to privileged instructions beyond its compiler attestation to the lack of inline assembly, meaning any privileged intrinsic or builtin is useable from inside of a CARAT KOP protected module. Instrumentation and wrappers to these builtins could be added during compilation, such that a guard is injected and a different policy table could be consulted to determine if a given kernel module has access to a privileged intrinsic. CARAT KOP also does not prevent control-flow attacks, where a module might call an arbitrary function in the kernel to perform a potentially malicious task. Incorporating guarded modules [12] into the CARAT KOP compilation flow would help CARAT KOP make assurances about control flow integrity of protected modules.

6 RELATED WORK

SFI and BGI. Compiler-based techniques to implement software protections is not a new concept. Perhaps the earliest work is Software Fault Isolation (SFI) [4, 10, 30, 35], which focused on user programs regardless of trust. Address sanitizers, for example EffectiveSan [7], are also closely related. Embedded operating systems such as Tock [19], Theseus [3], and RedLeaf [22] leverage the properties of Rust to build protection without hardware support. Unfortunately, these are not directly applicable to Linux kernel modules.

Byte-Granularity Isolation (BGI) [5] is of particular note. In contrast to SFI, BGI focuses on memory access rights to ensure protection, while SFI concentrates on enforcing code generation constraints to prevent faults. BGI offers a distinctive protection framework by creating separate protection domains within the same address space, each domain governed by Access Control Lists (ACLs) for each byte of virtual

memory. BGI's commitment to type safety, combined with its detection mechanisms for common domain-specific errors, works cohesively to reduce risks associated with faulty kernel extensions. This granularity, while providing a robust containment, can be resource-intensive, especially when every byte in memory requires an independent ACL. CARAT KOP's simpler policy scheme may suit caches better. Also unlike CARAT KOP, BGI does not apply ACLs to reads, raising the spectre of data-leakage or exfiltration.

Sandboxing, verification, and eBPF. Sandboxing, more specifically eBPF [1], is also a related area of work. The idea of this work is to set up an environment to run code in the kernel where it is isolated and can be observed. eBPF does this via a CARAT-like process, but it does not apply to arbitrary code. Rather, it runs on pseudo-C code that is aware of eBPF. Therefore, it does not apply directly to general Linux kernel modules as CARAT KOP does.

eBPF's primary advantage lies in its assurance that user-injected code is benign to the kernel *in general*—guaranteeing termination, safe memory access, and no information leakage to unprivileged users. However, eBPF's verifier—responsible for these guarantees—does not execute *formal verification*, which, combined with the complexity of its code and absence of a comprehensive testing mechanism, raises concerns about its infallibility. The verifier operates on an implicit blacklist principle, implying that for it to be effective, every potential attack vector should be previously known and neutralized—a significant expectation. The guarding mechanism in CARAT KOP aims to avoid the complexities of a blacklist approach.

Finally, eBPF has numerous restrictions, such as instruction limits and constraints on loops, that limit the possible use cases compared to general purpose kernel modules.

Proof-carrying code. Proof-carrying code (PCC) [24–26] is powerful way to allow safe kernel extensibility. If code can carry a verifiable proof with it that it is safe with respect to some security policy, and the proof can be quickly verified, then it is possible to eliminate all guards. Using PCC requires that a proof be constructed, however, and this is challenging for complex kernel modules.

Virtualization, driver domains, and LVD. Hardware virtualization can be used to isolate code. The closest work here is LVD [23]. LVD creates Lightweight Virtualized Domains (LVDs) by implementing a lightweight hypervisor responsible for moderating the use of privileged instructions, as well as switching nested page tables at isolation boundaries to control memory access. Each module runs under different guest-physical to host-physical address mapping (different nested page tables), and communication between modules and the kernel is done with with Lightweight Execution Domains (LXDs) [21] which make guarantees about preventing shared state that may break isolation. Entries to LVDs generate new per-CPU stacks. LVDs use of LXDs requires each module to have a specification for its interaction with the kernel, which is not simple to construct for many modules.

KSplit. KSplit [15] tries to partition device drivers (the most buggy modules, typically) from the core kernel. KSplit isolates device drivers through a set of LLVM passes and an IDL compiler. The driver module is analyzed to automatically identify shared state between the kernel and the driver, and the synchronization requirements it needs. It then generates code which automates such synchronization. In contrast, CARAT KOP focuses simply on guarding individual memory accesses in any kernel module. Though an undoubtedly more comprehensive solution for driver code, the simplicity of the guarding mechanism in CARAT KOP lends well other kinds of modules, and allows an operator to set a policy, much like a firewall.

7 CONCLUSION

Being able to deploy custom Linux kernel modules in HPC systems, including production systems, would provide many potential benefits, including the adoption of alternative or cutting-edge research ideas. However, this is impeded by legitimate safety concerns that stem from the monolithic nature of the Linux kernel. We have argued that it is possible to protect the core kernel from such kernel modules using a combination of compile-time and run-time techniques. CARAT KOP and the demonstration of using it to easily isolate/firewall a relatively large driver with only minimal performance overhead provide support for this claim.

There are a range of other approaches to isolating Linux kernel modules, and CARAT KOP could be itself be enhanced to provide more flexible and complex policies, as well as to provide more isolation. We hope that some combination of approaches will be adopted for the benefit of HPC users.

Acknowledgements

This work was made possible with support from the United States National Science Foundation (NSF) via grants CNS-1763743, CCF-2028851, CCF-2119069, CNS-2211315, and CNS-2211508, via the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

[1] [n. d.]. What is eBPF? an introduction and deep dive into the EBPF technology. https://ebpf.io/what-is-ebpf

[2] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[3] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1–19.

[4] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 45–58.

[5] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/1629575.1629581

[6] Peter Dinda, Nick Wanninger, Jiacheng Ma, Alex Bernat, Charles Bernat, Souradip Ghosh, Christopher Kraemer, and Yehya Elmasry. 2022. FPVM: Towards a Floating Point Virtual Machine. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3502181.3531469

[7] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*.

[8] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[9] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Institute of Electrical and Electronics Engineers, 1041–1050. https://doi.org/10.1109/IPDPS.2016.80

[10] GoogleNativeClient [n. d.]. Native Client. https://developer.chrome.com/native-client.

[11] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR]

[12] Kyle Hale and Peter Dinda. 2014. Guarded Modules: Adaptively Extending the VMM's Privileges Into the Guest. In *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 2014)*.

[13] Kyle Hale and Peter Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*.

[14] Kyle C. Hale and Peter A. Dinda. 2015. A Case for Transforming Parallel Runtimes Into Operating System Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/2749246.2749264

[15] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 613–631. https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe

[16] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. MOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 113–129.

[17] Brian Kocoloski and John Lange. 2014. HPMMAP: Lightweight Memory Management for Commodity Operating Systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Institute of Electrical and Electronics Engineers, 649–658. https://doi.org/10.1109/IPDPS.2014.73

[18] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. 2010. Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*.

[19] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 234–251.

[20] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. 2022. NOELLE Offers Empowering LLVM Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*

[21] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 269–284.

[22] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 21–39.

[23] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 157–171. https://doi.org/10.1145/3381052.3381328

[24] George Necula. 1997. Proof-carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 1997)*.

[25] George Necula and Peter Lee. 1996. *Proof-Carrying Code.* Technical Report CMU-CS-96-165. School of Computer Science, Carnegie Mellon University.

[26] George Necula and Peter Lee. 1996. Safe Kernel Extensions Without Run-time Checking. In *Proceedings of the 2nd USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 1996).

[27] Jiannan Oayang, Brian Kocoloski, John Lange, and Kevin Pedretti. 2015. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th ACM International Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)*.

[28] Swann Perarnau, Judicael A. Zounmevo, Matthieu Dreher, Brian C. Van Essen, Roberto Gioiosa, Kamil Iskra, Maya B. Gokhale, Kazutomo Yoshii, and Pete Beckman. 2017. Argo NodeOS: Toward Unified Resource Management for Exascale. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Institute of Electrical and Electronics Engineers, 153–162. https://doi.org/10.1109/IPDPS.2017.25

[29] 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. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1064–1079. https://doi.org/10.1145/3453483.3460969

[30] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).

[31] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 512–526.

[32] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/1294261.1294294

[33] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. 2020. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 329–345. https://doi.org/10.1145/3385412.3385987

[34] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 98–114. https://doi.org/10.1145/3503222.3507771

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP 1993)*.

[36] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proc. VLDB Endow.* 13, 2 (oct 2019), 197–210. https://doi.org/10.14778/3364324.3364333