Evolving Operating Systems Towards Secure Kernel-Driver Interfaces

Anton Burtsev
University of Utah

Vikram Narayanan
University of Utah

Yongzhe Huang
Pennsylvania State University

Kaiming Huang
Pennsylvania State University

Gang Tan
Pennsylvania State University

Trent Jaeger
Pennsylvania State University

Abstract

Our work explores the challenge of developing secure kernel-driver interfaces designed to protect the kernel from isolated kernel extensions. We first analyze a range of possible attack vectors that exist in current isolation frameworks. Then, we suggest a new approach to building secure isolation boundaries centered around ideas that originate in safe operating systems: isolation of heaps and single ownership.

CCS Concepts

• **Security and privacy → Operating systems security**.

ACM Reference Format:
Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. 2023. **Evolving Operating Systems Towards Secure Kernel-Driver Interfaces. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595914**

1 Introduction

A steady increase in the number of security attacks (combined with a growing level of attack complexity and automation) triggered a renewed interest in hardware support for isolation. After decades of relatively slow adoption, recent generations of commodity CPUs introduced a range of new hardware isolation mechanisms. Intel Memory Protection Keys (MPKs) develop support for memory isolation with overheads gradually approaching the overhead of a function call [16, 40]. The latest ARM CPUs introduce 16-byte-granularity memory isolation with the Memory Tagging Extensions (MTE) [2, 5], which can potentially enable low-overhead bounds checks and zero-copy exchange of data. Moreover, both ARM and x86 provide support for control flow integrity (CFI) [29] and stack protection [1, 28]. Finally, safe programming languages like Rust are becoming first-class citizens in modern systems [15, 43].

In response to lowering overheads of hardware and software isolation, a range of projects started to explore techniques for isolating legacy systems, many targeting isolation of the kernel subsystems like device drivers [9, 30, 31, 34, 36]. Furthermore, recent static analysis techniques demonstrated largely automated isolation of the kernel code [17]. With breakthroughs addressing two key isolation challenges—performance and complexity—it is likely that isolation will soon find its way into modern kernels.

A natural question, however, is what kind of security guarantees are achieved by current isolation frameworks? Unfortunately, even using the most advanced isolation boundary approaches that enforce temporal memory safety across the isolation boundary [30, 31], the kernel can be attacked in numerous ways.

Our work explores a possibility to develop new secure kernel-driver interfaces designed to protect the kernel from isolated drivers. We first analyze a range of possible attack vectors that exist in current isolation frameworks. Then we suggest a new isolation boundary centered around two ideas that originate from safe operating systems [6, 18, 35, 41]: isolation of heaps and single ownership. These ideas allow us to enforce safety of access to the data structures exchanged with isolated subsystems (even if the driver is malicious) as well as provide clean termination and restart of isolated drivers in case of a crash.

2 Modern isolation mechanisms

Historically, x86 CPUs supported two isolation mechanisms: segmentation and paging. Segmentation was demonstrated as a low-overhead isolation mechanism by L4 microkernels [27] and by recent WebAssemby implementations [22]. Moving to the 64-bit mode of execution, x86 CPUs deprecated segmentation, leaving paging as the only isolation

HotOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595914

mechanism available in x86 CPUs. Despite many optimizations, paging remains prohibitively expensive for isolation of subsystems that require frequent communication.

Trying to improve isolation overheads, CPU vendors explore the space of isolation mechanisms and introduce support for lightweight in-process isolation, control flow integrity (CFI), and software fault isolation (SFI).

Intel MPK Starting with SkyLake, Intel supports memory protection keys (MPK), which enforce isolation within a single address space by tagging individual pages with a 4-bit tag (saved in the unused bits of the page table entries). A special register, pkru, holds a bitmap that allows access to a combination of tags (i.e., any combination from none to all is possible by setting individual bits). The read or write access to a page is allowed only if the value of the pkru register matches the tag of the page. Switching between isolated subsystems is performed by writing a new tag value into the pkru register with an unprivileged wrpkru instruction that takes 20-26 cycles [16, 37]. The total cost of a cross-subsystem invocation is higher as it requires zeroing out general and extended registers, and choosing a new stack inside the caller subsystem [26]. These overheads can be lowered through additional guarantees on the structure of isolated code [22].

ARM MTE Starting with ARMv8.3-A, ARM SoCs introduce support for memory tagging extensions (MTE) that allow partitioning the address space into 16-byte regions that are colored with one of 16 tags. The hardware maintains a table that stores a mapping between addresses and tags allowing access to the region only if the tag of the pointer (the tag is stored in the upper bits of the pointer) matches the tag of the memory region. MTE itself does not directly support isolation – the attacker can change the upper bits of the pointer that contain the tag. To enforce isolation, it is possible to combine MTE with techniques of software fault isolation (SFI), i.e., rely on compile-time instrumentation to enforce a specific tag on every load and store operation.

ARM PAC Along with MTE, ARMv8.3-A introduced support for cryptographic pointer authentication (PAC). PAC implements the ability to cryptographically sign a pointer and store the signature in the "unused" upper bits of the pointer. The signature is generated from 1) the pointer value, 2) a secret key protected by the operating system, and 3) a 64-bit program-defined "signing context" that allows the isolation scheme to restrict the use of a pointer in a custom way, for example, allowing use of the pointer only if the value of the stack pointer is identical at signing and authenticating the signature. A signed pointer cannot be used directly, but instead has to be authenticated with the same secret key and context. If either the pointer, its signature, or the context is different from the values used during signing, the authentication results in an invalid pointer value that triggers a hardware exception when used. PAC is a powerful mechanism that can be used to enforce control flow [28], spatial [29] and temporal [14, 25] safety and isolation of subsystems [31]. Isolation with PAC requires maintaining metadata about each memory object, i.e., the size, type, and liveness of an object that is used to enforce type, memory, temporal safety, and the access rights for a currently executing isolated subsystem.

Rust Language safety allows enforcing isolation through rules of spatial and temporal safety combined with the access control model of the language (e.g., visibility of public and private fields and modules). Historically, the overhead of managed runtimes was prohibitive for low-level systems (Emmerich et al. provide a performance comparison of ten different languages used for a development of a fast network device driver [12]). A number of research prototypes explored language-based isolation [6, 7, 18, 42], but it remained impractical due to the overhead of managed language runtimes (traditionally, safe languages rely on garbage collection to enforce safety). Rust enabled practical, near zero-cost language safety through a restricted ownership model that allows ensuring safety without garbage collection. Today, Rust is accepted into both Linux and Windows kernels [15, 43].

CHERI Morello Hardware capability pointers describe the lower and upper bounds of a memory location and recursive pointer fields [44]. Effectively, hardware capabilities implement memory safety in hardware [10]. Two main benefits of hardware capability architectures are ability to propagate metadata along with pointers and low-overhead metadata checks. ARM Morello is the first silicon implementation of the CHERI architecture [3].

3 Attacks across isolation boundaries

Isolation frameworks Historically, isolation frameworks utilized the following mechanisms to enforce isolation: hardware isolation primitives, software fault isolation (SFI), and programming language safety. Hardware-based approaches execute an isolated subsystem, e.g., a device driver, on a private, isolated copy of all data structures shared between the driver and non-isolated kernel [36, 39]. Shared state is synchronized on cross-subsystem invocations and around synchronization primitives. Code annotations [39] or a general interface definition language (IDL) [17, 34, 36] specifies the access rights for the fields exchanged across subsystems. An attacker that has a write primitive inside the isolated subsystem cannot access the state of the kernel, but can modify any of the shared fields hence affecting the kernel.

SFI-based approaches execute the driver and the kernel on a single copy of the shared state [9, 13, 30, 31] and instead enforce access control on every memory access limiting the isolated subsystem to a set of allowed data structures and their fields. This eliminates the need for maintaining two copies of the shared state, but requires frequent access-control checks (i.e., on every memory access) [30, 32]. Compared to hardware-based approaches, SFI solutions enforce control-flow integrity, yet allowing an attacker to leverage control flow bending [8] and automated data-only attacks [19, 45] possibly giving them access to the same set of data structures and fields shared with the isolated subsystem. SFI techniques enforce a subset of memory safety; e.g., HAKC [31] enforces spatial and type safety, but not temporal safety.

Finally, while being impractical for decades due to the overheads of managed runtime, Rust is now used for development of subsystems in modern commodity kernels [15, 43]. Rust (and specifically its safe subset) significantly limits the space of possible low-level attacks. Yet even subsystems implemented in safe Rust can have code that breaks high-level invariants of the kernel that result in unsafe accesses in the kernel, missing security checks, denial of service and more.

In general, kernel and drivers share the entire address space, i.e., drivers have access to complex, hierarchical data structures inside the kernel through a collection of driver and helping kernel functions. In practice, each driver accesses only a subset of recursively reachable data structures and their fields. These fields can be pointers to other data structures and scalar (non-pointer) fields. Many of the non-pointer fields, however, have complex semantics, e.g., can be involved in allocation operations (e.g., define the size and number of allocated memory regions), participate in pointer arithmetic (e.g., define the size of dynamically-sized objects), specify types (e.g., the kernel uses tagged unions and other schemes to implement polymorphism), control liveness of memory objects and interfaces through state flags and reference counting, trigger protocol transitions, and more. Even if the isolated subsystem is safe, modification of these fields can be unsafe in the kernel.

Below, we discuss a range of attacks that are possible even with the strongest isolation schemes that enforce control-flow integrity and memory safety inside the isolated subsystem. While enforcing safety inside the isolated subsystem, a typical isolation solution leaves non-isolated kernel unprotected [30, 31]. We assume that in general isolated drivers are not malicious, but, since they are developed by third-party maintainers that have only partial understanding of the kernel security idioms, they might have arbitrary errors in their implementation. We use the Linux kernel for our examples although arguably other commodity kernels are subject to similar attacks.

Memory bounds An attacker can trigger an out-of-bounds access through the fields that control offsets into memory regions and their sizes. The accesses to non-pointer fields are safe inside the isolated driver, but can trigger unsafe behavior inside the kernel. For example, the `skb` data structure that describes a network packet in the kernel contains scalar fields, `tail` and `end`, that are used as offsets into the data buffer and are used as pointers by the kernel. An accidental update of the `tail` offset past the allocated `data` memory will result in an out-of-bounds access in the kernel.

Pointer aliases While typically an attacker cannot break type safety of a pointer (i.e., assign a pointer to an object of a different type), they can confuse the kernel and trigger a use-after-free or double free by creating duplicate aliases to the same object.

```
skb1->data = skb2->data;
```

When the kernel frees `skb1` it deallocates its `data` area, hence making the `data` pointer dangling in `skb2`, which results in a use-after-free or a double free.

Function pointers To implement interfaces, Linux relies on function pointers. An isolated driver can initialize a function pointers of the same type but incompatible semantics. In the example below, the driver initializes the `.ndo_open` field of the network interface with the `netdev_reset_tc()` function instead of `ixgbe_open()` (functions have identical type).

```
struct net_device_ops netdev_ops = {
  // .ndo_open = ixgbe_open,
  .ndo_open = netdev_reset_tc,
}
```

The function mismatch can result in a protocol violation confusing the kernel in multiple ways.

Lifetimes (use-after-free and double free) While it is possible to enforce temporal safety inside an isolated subsystem, the interface of a kernel leaves opportunities for breaking lifetimes of objects on the kernel side. For example, an isolated driver can invoke `free_netdev(dev)` to deallocate the `dev` data structure that is supposed to be live until the driver is unloaded by the kernel. Similarly, for reference counted objects like `skb`, the driver can drop the reference to the same `skb` structure twice by invoking the `consume_skb(skb)` function twice. Since additional references might exist in the kernel, there is a possibility of a use-after-free inside the kernel.

Resource exhaustion The kernel's use of data from isolated subsystems leaves plenty of opportunities for resource exhaustion attacks. The attacks range from simple requests for more memory to more sophisticated examples in which an isolated subsystem can trigger allocation of an object on the kernel side though one of the available interfaces, or refuse to deallocate one of the resoures that it is supposed to deallocate asynchronously.

Denial of service Denial of service attacks can range from a simple attack in which the subsystem never returns to more sophisticated attacks in which an isolated subsystem returns an error value that triggers shutdown of the driver.

Synchronization and consistency A range of attacks can trigger a deadlock by acquiring and never releasing a lock or breaking consistency guarantees about data structures by accessing them without a lock. For example, the network subsystem in the kernel allows any device driver to acquire a global RTNL kernel lock, hence blocking the execution of the entire network subsystem.

Protocol violations A driver can confuse the kernel by violating one of implicit initialization or operation protocols, i.e., invoking kernel functions out of order. In fact, nearly every shared scalar and some shared pointer fields control the logic of the kernel enabling and disabling functionality, triggering allocation and deallocation of objects, specifying types and sizes of objects, etc.

Security checks In some cases drivers are responsible for performing security checks.

```
if (!capable(CAP_SYS_RAWIO))
  return -EPERM;
```

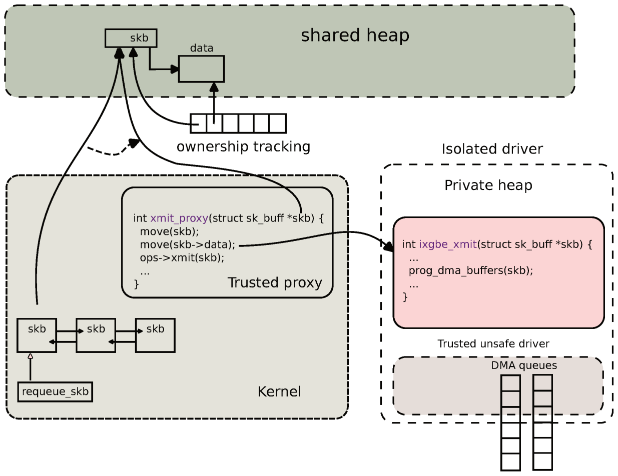
The driver can fail to implement the capability check to allow a user process to access an otherwise inaccessible resource.

Unrestricted hardware access Device drivers can access any hardware and software resources of the system. For example, a character driver can register or unregister a character device for any major or minor number. In case of an isolated MSR driver, the driver has unrestricted access to write to any of the MSR registers.

Discussion Even advanced safety and control-flow enforcement mechanisms fail to block the attacks exploiting a logical flow in the driver. While arguably some of the above attacks can be fixed with static code analysis, e.g., enforcing specific instances of the function pointers in driver interfaces, checking that the driver performs a security check on the path of an invocation, etc., and some require techniques orthogonal to the interface of the driver, e.g., DMA protection [4], a range of attacks can only be stopped through a redesign of a driver-kernel interface. Such interface should enforce a range of restrictions on pointer references exchanged across the isolation boundary, control object lifetimes, and provide fine-grained access control to hardware and systems resources.

4 Towards secure boundaries

Securing complex, semantically-rich interfaces of the kernel is challenging. We leverage ideas from safe operating systems [6, 18, 35, 41]—isolation of heaps and single ownership—a discipline that creates a foundation for clean termination and restart of isolated subsystems in case of a crash. We then suggest several ways of enforcing safety of access to the data structures exchanged with the driver. Finally, we adapt the idea of interfaces, or traits, as a way to implement object capability proxy pattern [11, 33], i.e., mediate access to hardware resources and data structures in the kernel. Fine-grained control over what kernel functions and with what arguments can be invoked at any given moment of driver's execution allows us to limit access to a specific subset of software and hardware resources of the kernel and enforce a specific protocol on the driver interface.



```
int xmit_proxy(struct sk_buff *skb) {
  move(skb);
  move(skb->data);
  ops->xmit(skb);
  ...
}
```

```
int ixgbe_xmit(struct sk_buff *skb) {
  ...
  prog_dma_buffers(skb);
  ...
}
```

Figure 1: We split the shared skb into private and shared parts and only the shared part is passed across the isolation boundary. The ownership of these shared objects are moved to the destination domain via trusted proxies.

Heap isolation We execute isolated device drivers as isolated subsystems with private heaps and a special shared *exchange heap*—a heap that allows allocation of objects that can be exchanged between the main kernel and isolated subsystems. Clean separation of private and exchange heaps allows us to enforce safety of accesses to objects exchanged with isolated subsystems as well as support clean termination semantics. In other words, while accesses to the private heap inside the isolated subsystem can be unsafe, they only affect the isolated subsystem itself. The isolated subsystem cannot break spatial or temporal safety of objects on the shared heap, hence all accesses to the shared heap from the kernel are safe.

Fault isolation Safety is critical to enforce confidentiality and integrity of the kernel, i.e., protect it from low-level vulnerabilities. However, safety alone is insufficient for making progress in case of a crash; when an isolated driver crashes it leaves its heap and objects exchanged with the kernel in an inconsistent state. Cleaning the state of the driver and especially cleaning the objects that are shared with the kernel is challenging. To support clean termination and unloading of crashing subsystems, we enforce several additional invariants: 1) **Inv 1**: *heap isolation* – subsystems never hold pointers into each others' private heaps, 2) **Inv 2**: *single ownership* – objects on the shared heap are owned and can be accessed by exactly one subsystem at a time and are moved between them on cross-subsystem invocations, 3) **Inv 3**: *linearity* – objects on the shared heap do not have cyclic references.

The IPC subsystem tracks ownership of objects on the shared exchange heap (in the following sections we suggest several ways of enforcing single ownership and providing ownership tracking specific to each isolation mechanism). When a subsystem crashes, the kernel deallocates all objects on the shared heap owned by the crashing subsystem as well as deallocates its shared heap. Due to **Inv 1** we can deallocate the entire heap without worrying about its state (we only track pages used by the private heap). **Inv 2** guarantees that we can deallocate objects owned by the crashing subsystem without worrying about accesses from other subsystems (and hence without implementing reference counting or garbage collection mechanisms). Finally, **Inv 3** allows us to enforce the safety of accesses to the shared heap. Linearity strikes a balance between ergonomics and security. Specifically, the kernel can exchange hierarchical data structures (not just plain buffers), and yet we can eliminate aliasing attacks.

On cross-subsystem invocations, the IPC subsystem records the state of the caller thread upon entry into the callee subsystem, thus allowing us to unwind execution of a thread from the crashing subsystem.

4.1 Safety in the kernel

While safety and heap isolation are appealing, the real question is whether it is possible to adapt them in the kernel in a practical manner? We argue that there are several ways to enforce safety of the shared heap by either rewriting isolated device drivers in a safe language like Rust, or by relying on modern software fault isolation (SFI) mechanisms [25, 30].

Rust Rust is a programming language that enforces safety through a restricted ownership discipline allowing only one mutable reference to each live object in memory [21]. The compiler leverages restricted ownership to reason about lifetimes statically at compile time. Without a managed runtime Rust approaches the performance of unsafe programming languages. Rust's type system enforces safety over the objects on the shared heap. Moreover, the restricted ownership model enforces linearity.

Note, device drivers are inherently unsafe due to the fact that they access an unsafe hardware interface. Unsafe Rust breaks all safety guarantees similar to any unsafe language. Fortunately, it is possible to implement a large fraction of the driver in a safe subset of Rust and rely on a small unsafe library that encapsulates unsafe mechanisms required to access the hardware [24, 35]. While this unsafe library becomes a part of the TCB, its semantics are simple enough that it can be verified with modern verification tools [20, 23].

Temporal safety with PAC Alternatively, safety of the objects on the shared heap can be enforced through a combination of modern software fault isolation (SFI) mechanisms [25, 29, 30] or hardware capabilities like Cheri [44]. Systems like LXFI [30], PARTS [29], and PACMem enforce checks on all memory accesses to ensure types, bounds, and liveness of objects (i.e., spatial and temporal safety). Specifically, PACMem relies on ARM pointer authentication to implement efficient metadata lookups (PACMem uses PAC signature of a pointer as an index into the metadata table) [25]. To further improve performance of the isolated drivers, it is possible to leverage **Inv 1** and enforce PAC memory safety only on pointers to the exchange heap. An SFI scheme first checks if the pointer is local and if so simply ensures that it is in bounds similar to fast SFI implementations [38]; otherwise it resorts to a full safety check through the metadata. Similar to Rust, an unsafe layer is required to encapsulate unsafe hardware interface of the driver.

Single ownership and linearity Rust enforces single ownership (and hence linearity) through a restricted ownership discipline. If the device driver is implemented in safe Rust, we automatically have guarantees of single ownership inside the driver (i.e., after passing a pointer to an object on the shared heap, the driver can no longer access the object). Similarly, by restricting the types that are allowed on the shared heap [35] we can enforce acyclic data structures on the exchange heap.

While re-writing device drivers in Rust is a laudable idea, majority of kernel device drivers will remain implemented in C for years to come. At the moment, we do not see a way of enforcing single ownership in unsafe C due to unrestricted pointer aliasing. Potentially, it is possible to combine PAC-style safety to restrict aliasing dynamically with static and dynamic ownership analysis. However, subsystem-wide safety enforcement introduces high overhead due to expensive metadata lookups [25]. Instead, for legacy drivers we fall back to a practical scheme that checks safety and linearity only on invocations that cross the isolation boundary. Specifically, for each invocation we leverage an IPC declaration that explicitly defines object hierarchy passed across subsystems [17, 36]. Automatically generated IPC code walks the hierarchy, checking safety and linearity along with enforcing access control rules; i.e., all objects are owned by the subsystem that passes them while at the same time transferring ownership and revoking access rights. Depending on the enforcement mechanism we either update the memory tag (MTE or MPK) or access metadata similar to LXFI [30].

4.2 Bringing linearity to the kernel

While it is possible to enforce safety and single-ownership on the isolation boundary, an obvious question is whether it is possible to integrate linearity with existing kernel code without disruptive changes? We make a critical observation that while data structures shared with the driver are used in a non-linear manner, a subset of fields and memory objects that are accessed by the driver is small and can be made linear. For example, the `sk_buff` data structure is used in a complex non-linear manner by the kernel which keeps `sk_buff`s on multiple linked lists for retransmission of network packets, auditing frameworks, etc. However, only a small subset of the `sk_buff` fields and objects reachable from these fields are accessed by the driver. Out of 66 fields of the `skb` data structure only 20 are used by the `ndo_start_xmit()` function of the Ixgbe device driver.

We split the `skb` into the private (used by the kernel) and shared (exchanged with the driver) parts. We then change the kernel to access the shared part of the `skb` in a linear manner. Specifically, the `xmit_proxy` function moves ownership of the private part along with the `data` region that holds the actual packet data to the driver and then `consume_skb` moves back to the kernel when DMA transfer is complete.

Interface proxying In many cases the kernel exchanges references with the driver to data structures that are mostly private to the kernel. The driver accesses a small number of fields (in many cases the pointer is not even used by the driver but only passed in nested invocations to the kernel). Such accesses are often not linear as the kernel accesses the same object from a variety of kernel contexts, e.g., interrupts, kernel threads and system call invocations. For instance, a typical network driver has access to the following kernel objects: `pci_dev` (generic PCI device), `pci_dev->dev` (generic representation of a device), and `netdevice` (generic network device).

To ensure linearity, instead of sharing a reference to the kernel object with the driver, we allow the driver to access an interface of a specific object through a proxy interface. Proxies implement an object capability pattern [11, 33], effectively providing an access to a subset of fields in a controlled manner, and even enforcing a specific access protocol, e.g., order of proxy invocations, invoke only once, etc. In Rust we implement proxy objects as Rust traits. For example, the PCI trait allows the network device driver to access the configuration space of the PCI from the probe function.

```
pub trait PCI {
  fn read_config_dword(where: u32) -> u32;
  fn write_config_dword(where: u32, val: u32) -> i32;
}

fn ixgbe_probe(pci: &PCI) {
  ...
  val = pci.read_config_dword(where);
  ...
}
```

For legacy C drivers we leverage LXDs' support for exchanging function pointers across isolation boundaries [34].

Discussion Safety and linearity of the shared heap allow us to protect the kernel from a range of pointer integrity attacks. We leverage our prior work on KSplit [17] to develop static analysis algorithms that allow us to identify scalar fields involved in pointer arithmetic, bounds computations, etc. We then utilize KSplit's IDL to encode and generate proxy invocation code that sanitizes the risky fields.

Protocol violations is arguably the broadest class of attacks on the kernel due to its semantic complexity. We developed a collection of static analyses that allow us to reason about how scalar fields that are controlled by the driver affect execution of the kernel, i.e., trigger allocation and deallocation of objects, enable functionality, etc. We then leverage KSplit's IDL to encode kernel-driver interaction protocols and generate enforcement through proxy trait objects. Moreover, we rely on the interface interposition to enforce specific ordering of function invocations. Specifically, we encode a state machine of a driver-kernel protocol and allow invocations of the functions that are allowed by transitions of the protocol state machine.

We further leverage trait objects to enforce fine-grained access control on access to resources of the system. By wrapping the access to specific objects behind interface pointers, we provide isolated drivers with the mechanism to invoke kernel functions but with a predefined subset of arguments, i.e., limiting the driver to a specific subset of the PCI space, or limiting an MSR driver to a write of a specific MSR register.

To prevent synchronization attacks, we forbid critical sections that cross the boundaries of isolated subsystems. Instead the driver has to call out into the kernel to synchronize (fortunately, the cases when synchronization patterns cross boundary of subsystems are rare in the kernel [17]).

5 Status

Our framework is a work in progress aimed at providing secure extensions in the Linux kernel. To aid decomposition of data structures into shared and private parts, we developed a static analysis framework that computes the parts of the data structure accessed by the driver and an interface definition language (IDL) compiler that generates IPC bindings and trait objects for Rust and C device drivers. Our static analysis utilizes a recent kernel isolation framework, KSplit [17].

6 Conclusions

Our work explores the possibility of isolating the core part of the kernel from third-party device drivers. We believe that a combination of heap isolation and single ownership provides a viable design choice for enforcing security of the isolation boundary and ensuring clean recovery from driver crashes.

Acknowledgments

We would like to thank HotOS 2023 reviewers for numerous suggestions helping us to improve this work. This research is supported in part by the National Science Foundation under grant numbers OAC-11840197, CCF-1837127, CNS-1816282 and CNS-1801534, and DARPA under HR0011-19-C-0106.

References

[1] A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[2] Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety.

[3] Morello research program hits major milestone with hardware now available for testing. https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing.

[4] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 395–409, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Arm. Armv8.5-A Memory Tagging Extension. *Whitepaper.* https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[6] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–283, 1995.

[8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 161–176, 2015.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 45–58, 2009.

[10] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130. ACM, 2015.

[11] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP 2013*, 2013.

[12] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[13] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88, 2006.

[14] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[15] Alex Gaynor and Geoffrey Thomas. Linux kernel modules in Rust. *Proceedings of the Linux Security Summit North America*, 2019, 2019.

[16] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 489–504, 2019.

[17] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

[18] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[19] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, New York, NY, USA, 2018. ACM.

[20] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[21] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, 2019.

[22] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: near-zero-cost transitions for webassembly and sfi. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30, 2022.

[23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

[24] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 234–251, 2017.

[25] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 1901–1915, 2022.

[26] Zhaofeng Li, Tianjiao Huang, Vikram Narayanan, and Anton Burtsev. Understanding the overheads of hardware and language-based ipc mechanisms. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, pages 53–61, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Jochen Liedtke. Improved address-space switching on pentium processors by transparently multiplexing user address spaces. Technical report, GMD SET-RS, Schlo Birlinghoven, 53754 Sankt Augustin, Germany, 1995.

[28] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374. USENIX Association, August 2021.

[29] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194. USENIX Association, August 2019.

[30] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 115–128, 2011.

[31] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[32] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[33] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

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

[35] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[36] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, pages 157–171, 2020.

[37] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, July 2019.

[38] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*, pages 1–11, 2010.

[39] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[40] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, 2019.

[41] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[42] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[43] David Weston. Windows 11. The journey towards default security. `https://www.youtube.com/watch?v=8T6ClX-y2AE`.

[44] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[45] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 781–797, 2018.