LXDs: Towards Isolation of Kernel Subsystems

Vikram Narayanan, ***University of California, Irvine;*** **Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, and Michael Quigley,** ***University of Utah;*** **Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev,** ***University of California, Irvine***

https://www.usenix.org/conference/atc19/presentation/narayanan

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

LXDs: Towards Isolation of Kernel Subsystems

Vikram Narayanan
University of California, Irvine

Abhiram Balasubramanian*
University of Utah

Charlie Jacobsen*
University of Utah

Sarah Spall†
University of Utah

Scott Bauer‡
University of Utah

Michael Quigley§
University of Utah

Aftab Hussain
University of California, Irvine

Abdullah Younis¶
University of California, Irvine

Junjie Shen
University of California, Irvine

Moinak Bhattacharyya
University of California, Irvine

Anton Burtsev
University of California, Irvine

Abstract

Modern operating systems are monolithic. Today, however, lack of isolation is one of the main factors undermining security of the kernel. Inherent complexity of the kernel code and rapid development pace combined with the use of unsafe, low-level programming language results in a steady stream of errors. Even after decades of efforts to make commodity kernels more secure, i.e., development of numerous static and dynamic approaches aimed to prevent exploitation of most common errors, several hundreds of serious kernel vulnerabilities are reported every year. Unfortunately, in a monolithic kernel a single exploitable vulnerability potentially provides an attacker with access to the entire kernel.

Modern kernels need isolation as a practical means of confining the effects of exploits to individual kernel subsystems. Historically, introducing isolation in the kernel is hard. First, commodity hardware interfaces provide no support for efficient, fine-grained isolation. Second, the complexity of a modern kernel prevents a naive decomposition effort. Our work on Lightweight Execution Domains (LXDs) takes a step towards enabling isolation in a full-featured operating system kernel. LXDs allow one to take an existing kernel subsystem and run it inside an isolated domain with minimal or no modifications and with a minimal overhead. We evaluate our approach by developing isolated versions of several performance-critical device drivers in the Linux kernel.

1 Introduction

Modern operating system kernels are fundamentally insecure. Due to rapid development rate (the de-facto industry standard Linux kernel features over 70 thousand commits a year), a huge codebase (the latest version of the Linux kernel contains over 17 million lines of unsafe C/C++ and assembly code[1]), and inherent complexity (typical kernel code adheres to multiple allocation, synchronization, access control, and object lifetime conventions) bugs and vulnerabilities are routinely introduced into the kernel code. In 2018, the Common Vulnerabilities and Exposures database lists 176 Linux kernel vulnerabilities that allow for privilege escalation, denial-of-service, and other exploits [20]. This number is the lowest across several years [19].

Even though a number of static and dynamic mechanisms have been invented to protect execution of the low-level kernel code, e.g., modern kernels deploy stack guards [18], address space layout randomization (ASLR) [45], and data execution prevention (DEP) [72], attackers come up with new ways to bypass these protection mechanisms [8, 35, 45, 49, 51, 57–59, 71]. Even advanced defense mechanisms that are yet to be deployed in mainstream kernels, e.g., code pointer integrity (CPI) [1, 53] and safe stacks [14, 53], become vulnerable in the face of data-only attacks combined with automated attack generation tools [46, 77]. In a monolithic kernel, a single vulnerability provides an attacker with access to the entire kernel. An attacker can redirect control to any part of the kernel and change any data structure escalating its privileges [46, 77].

Modern kernels need isolation as a practical means of confining the effects of individual vulnerabilities. However, introducing isolation in a kernel is hard. First, despite many advances in the architecture of modern CPUs, low-overhead hardware isolation mechanisms [74–76] did not make it into commodity architectures. On modern machines, the minimal call/reply invocation that relies on traditional address-spaces for isolation [26] takes over 834 cycles (Section 5). To put this number into perspective, in the Linux kernel a system call that sends a network packet through the network stack and network device driver takes 2299 cycles. A straightforward isolation of a network device driver which requires two domain crossings on the packet transmission path (Section 4), would introduce an overhead of more than 72%.

Second, the complexity of a shared-memory kernel that accumulates decades of development in a monolithic setting

*Currently at Ubiquiti Networks. Work done at the University of Utah.
†Currently at Indiana University. Work done at the University of Utah.
‡Currently at Qualcomm. Work done at the University of Utah.
§Currently at Google. Work done at the University of Utah.
¶Currently at the University of California, Berkeley. Work done at the University of California, Irvine
[1]Calculated using David Wheeler's sloccount on Linux 5.0-rc1.

prevents a trivial decomposition effort. Decomposition requires cutting through a number of tightly-connected, well-optimized subsystems that use rich interfaces and complex interaction patterns. Two straightforward isolation strategies—developing isolated subsystems from scratch [21, 24, 31] or running them inside a full copy of a virtualized kernel [11, 16, 30, 60]—result in either a prohibitively large engineering effort or overheads of running a full copy of a kernel for each isolated domain.

Our work on Lightweight Execution Domains (LXDs) takes a step towards enabling isolation in a full-featured operating system kernel. LXDs allow one to take an existing kernel subsystem and run it inside an isolated domain with minimal or no modifications and with a minimal overhead. While isolation of core kernel subsystems, e.g., a buffer cache layer, is beyond the scope of our work due to tight integration with the kernel (i.e., complex isolation boundary and frequent domain crossings), practical isolation of device drivers, which account for over 11 millions lines of unsafe kernel code and significant fraction of kernel exploits, is feasible.

Compared to prior isolation attempts [9, 12, 15, 27, 32–34, 40–43, 66, 68, 69], LXDs leverage several new design decisions. First, we make an observation that synchronous cross-domain invocations are prohibitively expensive. The only way to make isolation practical is to leverage asynchronous communication mechanisms that batch and pipeline multiple cross-domain invocations. Unfortunately, explicit management of asynchronous messages typically requires a clean-slate kernel implementation built for explicit message-passing [5, 42]. LXDs, however, aim to enable isolation in commodity OS kernels that are originally monolithic (commodity kernels accumulate decades of software engineering effort that is worth preserving). To introduce asynchronous communication primitives in the code of a legacy kernel, LXDs build on the ideas from asynchronous programming languages [3, 13, 39]. We develop a lightweight asynchronous runtime that allows us to create lightweight cooperative threads that may block on cross-domain invocations and hence implement batching and pipelining of cross-domain calls in a way transparent to the kernel code.

Second, to break the kernel apart in a manner that requires only minimal changes to the kernel code, we develop *decomposition patterns*, a collection of principles and mechanisms that allow decomposition of the monolithic kernel code. Specifically, we support decomposition of typical idioms used in the kernel code—exported functions, data structures passed by reference, function pointers, etc. To achieve such backward compatibility, decomposition patterns define a minimal runtime layer that hides isolated, share-nothing environment by synchronizing private copies of data structures, invoking functions across domain boundaries, implementing exchange of pointers to data structures and functions, handling dispatch of cross-domain function calls, etc. Further, to make our approach practical, we develop an interface definition language (IDL) that generates runtime glue-code code required for decomposition.

Finally, similar to existing projects [54, 67], we make an observation that on modern hardware cross-core communication via the cache coherence protocol is faster then crossing an isolation boundary on the same CPU. By placing isolated subsystems on different cores it is possible to reduce isolation costs. While dedicating cores for every isolated driver is impractical, the ability to run several performance-critical subsystems, e.g., NVMe block and network device drivers, with the lowest possible overhead makes sense.

We demonstrate practical isolation of several performance-critical device drivers in the Linux kernel: software-only network and NVMe block drivers, and a 10Gbps Intel ixgbe network driver. Our experience with decomposition patterns shows that majority of the decomposition effort can be done with no modification to the kernel source. We hope that our work—general decomposition patterns, interface definition language, and asynchronous execution primitives—will gradually enable kernels that employ fine-grained isolation as the first-class abstraction. At the moment, two main limitations of LXDs are 1) requirement of a dedicated core for each thread of an isolated driver (Section 5), and 2) manual development of the IDL interfaces. We expect to relax both of the limitations in our future work.

2 Background and Motivation

The concept of decomposing operating systems for isolation and security is not new [9, 12, 15, 27, 32–34, 40–43, 66, 68, 69]. In the past, multiple projects tried to make isolation practical in both microkernel [9, 27, 41–43, 68] and virtual machine [12, 15, 33, 66] systems. SawMill was a research effort performed by IBM aimed at building a decomposed Linux environment on top of the L4 microkernel [34]. SawMill was an ambitious effort that outlined many problems of fine-grained isolation in OS kernels. SawMill relied on a synchronous IPC mechanism and a simple execution model in which threads migrated between isolated domains. Unfortunately, the cost of a synchronous context switch more than doubled in terms of CPU cycles over the last two decades [26]. Arguably, with existing hardware mechanisms the choice of a synchronous IPC is not practical (on our hardware a bare-bone synchronous call/reply invocation takes over 834 cycles on a 2.6GHz Intel machine; a cache-coherent invocation between two cores of the same die takes only 448-533 cycles, moreover, this number can be reduced further with batching (Section 5)). Furthermore, relying on a generic Flick IDL [25], SawMill required re-implementation of OS subsystem interfaces. In contrast, LXDs's IDL is designed with an explicit goal of backward compatibility with the existing monolithic code, i.e., we develop mechanisms that allow us to transparently support decomposition of typical code patterns used in the kernel, e.g., registration of interfaces as function pointers, passing data structures by reference, etc.

Nooks further explored the idea of isolating device drivers in the Linux kernel [69]. Similar to SawMill, Nooks relied on the synchronous cross-domain procedure calls that are prohibitively expensive on modern hardware. Nooks maintained and synchronized private copies of kernel objects, however, the synchronization code had to be developed manually. Nooks' successors, Decaf [64] and Microdrivers [32] developed static analysis techniques to generate glue code directly from the kernel source. LXDs do not have a static analysis support at the moment. We, however, argue that IDL is still an important part of a decomposed architecture—IDL provides a generic intermediate representation that allows us to generate glue code for different isolation boundaries, e.g., cross-core invocations, address-space switches, etc.

OSKit developed a set of decomposed kernel subsystems out of which a full-featured OS kernel could be constructed [29]. While successful, OSKit was not a sustainable effort—decomposition glue code was developed manually, and required a massive engineering effort in order to provide compatibility with the interface of Component Object Model [17]. OSKit quickly became outdated and unsupported.

Rump kernels develop glue code that allows execution of unmodified subsystems of the NetBSD kernel in a variety of executable configurations on top of a minimal execution environment [48], e.g., as a library operating system re-composed out of Rump kernel subsystems. Rump's glue code follows the shape of the kernel subsystems and hence provides compatibility with unmodified kernel code that ensures maintainability of the project. LXDs follow Rump's design choice of ensuring backward compatibility with unmodified code, but aim at automating the decomposition effort. Specifically, LXDs rely on decomposition patterns and IDL to extract unmodified device drivers from the kernel source and seamlessly enable their functionality for the monolithic kernel.

User-level device drivers [11, 21, 24, 30, 60] allow execution of device drivers in isolation. Two general approaches are used for isolating the driver. First, it is possible to run an unmodified device driver on top of a device driver execution environment that provides a backward compatible interface of the kernel inside an isolated domain [24]. Unfortunately, development of a kernel-compatible device driver execution environment requires a large engineering effort. Sometimes, backward compatibility is sacrificed to simplify development, but in this case the device driver or a kernel subsystem have to be re-implemented from scratch [31]. LXDs aim to provide a general framework for automating development of custom backward compatible device driver environments. With a powerful IDL, fast communication primitives, and asynchronous threads, LXDs enable nearly transparent decomposition of kernel code.

Alternatively, the device driver environment is constructed from a partial or complete copy of the kernel that can host the isolated driver on top of a VMM [11, 30, 60] or inside a user process [11, 48]. Unfortunately, a virtualized kernel [11, 30, 60] extends the driver execution environment with a nested copy of multiple software layers, e.g., interrupt handling, thread scheduling, context-switching, memory management, etc. These layers introduce overheads of tens of thousands of cycles on the critical data-path of the isolated driver, and provide a large attack surface. A library operating system that provides full or partial compatibility with the original kernel can be used as an execution environment for the isolated device driver [48, 62, 70]. Smaller and lighter compared to the full kernel, library operating systems eliminate performance overheads of the full kernel. LXDs provide ability to run an unmodified device driver in a very minimal kernel environment hence achieving lean data path of a custom-built device driver execution environment.

3 LXDs Architecture

LXDs execute as a collection of isolated domains running side by side with the monolithic kernel (Figure 1). This design allows us to enable isolation incrementally, i.e., develop isolated device drivers one at a time, and seamlessly enable their functionality in the monolithic kernel.

Each LXD is developed as a loadable kernel module. An unmodified source of the isolated driver is linked against the two components that provide a backward compatible execution environment for the driver: 1) the glue code generated by the IDL compiler Figure 1, ⑥), and 2) a minimal library, libLXD (Figure 1, ⑦), that provides common utility functions normally available to the driver in a monolithic kernel, e.g., memory allocators, synchronization primitives, routines like `memcpy()`, etc.

LXDs rely on hardware-assisted virtualization (VT-x) for isolation. The choice of the hardware isolation mechanism is orthogonal to the LXDs architecture. VT-x, however, implements convenient interface for direct assignment of PCIe devices to isolated domains, and direct interrupt delivery (support for which we envision in the future). On the critical path LXDs rely on asynchronous cross-core communication primitives, and hence the cost of transitions to and from the VT-x domain (which is higher than a regular context switch) is acceptable.

LXDs are created and managed by a small microkernel that runs inside the commodity operating system kernel (Figure 1, ⑧). The LXD microkernel follows design of the L4 microkernel family [26]: it is centered around a pure capability-based synchronous IPC that explicitly controls authority of each isolated subsystem. The synchronous IPC is used for requesting microkernel resources, and exchange of capabilities, e.g., establishing regions of shared memory that are then used for fast asynchronous channels. Each LXD starts with at least one synchronous IPC channel that allows the LXD to gain more capabilities, exchange capabilities to its memory pages with the non-isolated kernel, and establish fast asynchronous communication channels.

To provide an interface of the isolated driver inside the

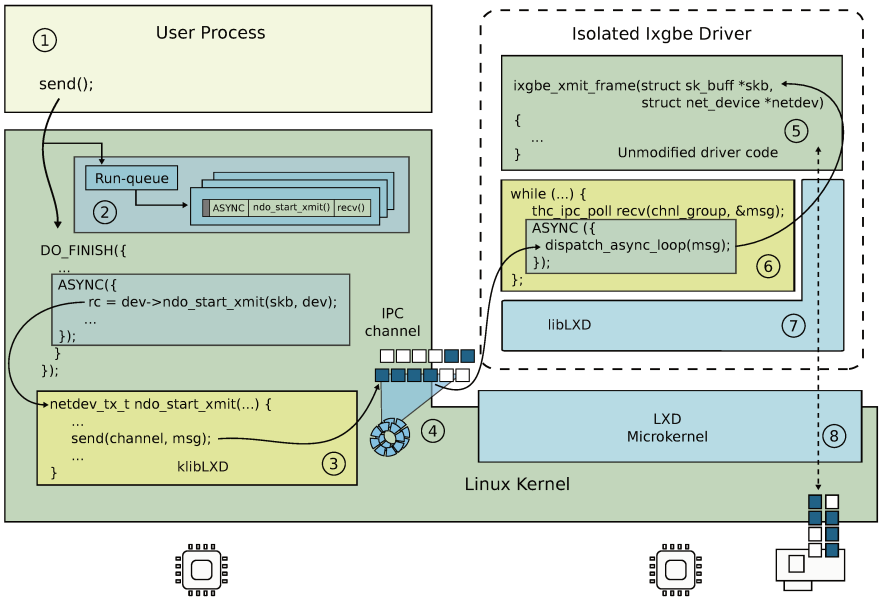

Figure 1: LXDs architecture (isolated ixgbe network driver).

monolithic kernel, every LXD loads the corresponding klibLXD module (Figure 1, ③). The klibLXD is automatically generated by the IDL compiler. The glue code inside klibLXD transparently marshals arguments of cross-domain invocations to the actual isolated driver.

In the example of the isolated ixgbe driver (Figure 1), a user process invokes an unmodified send() system call initiating transmission of the network packet through the isolated device driver (Figure 1, ①). The monolithic kernel relies on the interface of the isolated ixgbe driver (a collection of function pointers registered by the driver with the kernel) to pass the packet to the device (dev−>ndo_start_xmit()). The dev−>ndo_start_xmit() function pointer is implemented by the glue code of the klibLXD module. Internally, the glue code relies on the low-level send and receive primitives of the asynchronous communication channels to send the message to the isolated driver. The message reaches the isolated driver where it is processed by the dispatch loop generated by the IDL compiler (Figure 1, ⑥). The dispatch loop then invokes the actual ixgbe_xmit_frame() function of the unmodified ixgbe driver (Figure 1, ⑤).

Transparent decomposition LXDs rely on a collection of decomposition patterns to break the code of a monolithic system and emulate a shared-memory environment for isolated subsystems (Section 4). In LXDs, isolated subsystems do not share any state that might break isolation guarantees, e.g., memory pointers, indexes into memory buffers, etc. Instead, each isolated subsystem maintains its own private hierarchy of data structures. LXDs rely on a powerful IDL to automatically generate all inter-domain communication and synchronization code (Figure 1, ③ and ⑥). In contrast to existing IDLs used for constructing multi-server [25, 38] and distributed systems [23, 50, 61, 73] the main design goal behind the LXDs' IDL is backward compatibility with unmodified code. The IDL is designed to generate caller and callee stubs that hide details of inter-domain communication and synchronization of data structures.

Asynchronous runtime Compared to the monolithic kernel, a decomposed environment requires a cross-domain invocation in place of a regular procedure call for every function that crosses the boundary of an isolated domain. On modern hardware the overhead of such crossings is prohibitively expensive. LXDs include a minimal runtime built around lightweight asynchronous threads that aims to hide overheads of cross-domain invocations by exploiting available request parallelism. Specifically, the ASYNC() primitive creates a lightweight cooperative thread that yields execution to the next thread when blocked on the reply from an isolated domain (Figure 1, ②). Asynchronous threads allow us to introduce asynchrony to the kernel code in a transparent manner.

Cross-core IPC To reduce overheads of crossing domain boundaries, LXDs schedule isolated subsystems with tight latency and throughput requirements, i.e., network and block device drivers, on separate CPU cores. The reason is that on modern hardware cross-core communication via cache coherence is faster than a context switch on the same CPU. LXDs rely on efficient cross-core communication channels to send messages across isolated subsystems (Figure 1, ④).

3.1 Interface Definition Language

We develop a collection of *decomposition patterns*— a collection of principles and mechanisms, e.g., remote references, projections, and hidden arguments, that allow isolation of typical code patterns used in the kernel, e.g. exported functions, data structures passed by reference, registration of interfaces

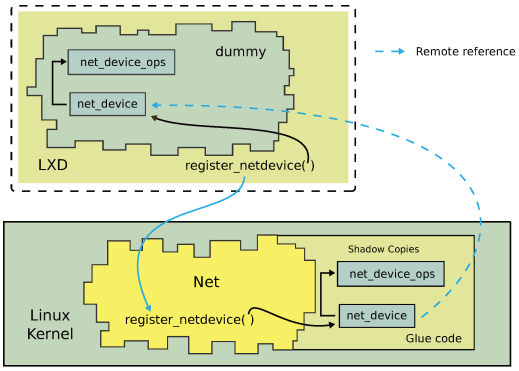

Figure 2: Private object hierarchies.

as function pointers, etc. To support implementation of decomposition patterns, we develop a powerful IDL that generates all inter-domain communication code.

Modules The IDL describes each subsystem as a *module*, i.e., a collection of functions exported and imported by an isolated driver or the kernel. To illustrate decomposition patterns and the design of the IDL, we consider an example of a minimal dummy network device driver [44]. The following IDL is used to define the dummy module.

```
include <net.idl>
module dummy() {
    require net;
}
```

By itself the dummy module does not export any functions. Instead it relies on the net interface provided by the kernel to register itself with the kernel, i.e., register a collection of function pointers that provide the driver-specific implementation of the network device interface. The kernel uses these function pointers to invoke the isolated dummy device driver.

The require keyword instructs the IDL compiler to import the net module into the context of the dummy module. At a high level, the compiler is instructed to generate glue code required for remote invocations of the functions exported by the net module.

A typical network interface defines a collection of functions that implement a specific kernel interface. For example, the net module defines the interface of the network subsystem, i.e., a collection of functions that allow network device drivers to register with the kernel.

```
module net() {
  rpc int register_netdevice(projection net_device *dev);
  rpc void ether_setup(projection net_device *dev);
  ...
}
```

From the above module definition the IDL generates code for caller stubs of the net interface so the isolated dummy module can transparently invoke functions of the interface. The IDL also generates the dispatch loops for both the dummy LXD and kLXD so both isolated subsystem and non-isolated kernel can process remote function invocations from each other.

Data structures In LXDs, isolated device drivers and the kernel do not share any state that might break isolation guarantees. Instead, each isolated subsystem maintains its own private hierarchy of data structures. In our example, the register_netdev() function takes a pointer to the net_device data structure that describes the network device. Since net_device is allocated inside the isolated dummy driver, a corresponding shadow copy will be created by the glue code in the non-isolated kernel (Figure 2).

The shadow hierarchies are synchronized upon function invocations. LXDs provide support for transparent synchronization of shadow data structure copies across domains with the mechanism of *projections*. A projection explicitly defines a subset of fields of the data structure that will be passed to the callee and returned to the caller during the domain invocation.[2]

```
projection <struct net_device> net_device {
  unsigned int flags;
  unsigned int priv_flags;
  ...
  projection net_device_ops [alloc(caller)] *netdev_ops;
}
```

Here, the projection net_device only lists the fields that will be used by the non-decomposed code in the kernel to register a network device. The projection omits the members of struct net_device that are private to the LXD, e.g., pointers to other data structures. The IDL supports lexical scopes, so the same data structure can be projected differently by different functions.

The IDL supports explicit [in] and [out] directional attributes to specify whether each field is marshalled from caller to callee or vice versa. In most cases, however, they are optional. The IDL compiler can infer the default direction from the way the projection is used in the code. In the example above, the default direction is [in]—all fields of the projection are copied from the caller to the callee side, which is decided based on the [alloc(callee)] qualifier that we discuss below.

Allocation of shadow object copies When the register_netdev() function is invoked by the LXD, the callee side of the invocation, i.e., the non-decomposed kernel, does not have a private version of the net_device data structure. The IDL provides support for controlling when remote objects are allocated, looked up, and freed with the alloc, bind, and dealloc qualifiers. The alloc qualifier instructs the IDL to allocate the new data structure of the projected type, i.e., struct net_device. The callee attribute instructs the IDL to perform the allocation on the callee side, as the data structure already exists on the caller side. The allocation attribute also serves as a hint to the compiler to marshal all fields of the projection from the already existing data

[2]Hence defining how a data structure is projected into another domain.

structure on the caller side to the callee (i.e., all fields of the projection above have the implicit [in] attribute). The data structure is deallocated when the dealloc qualifier is used with the projection.

Remote object references In most cases isolated subsystems refer to the same data structure multiple times. For example, the net_device data structure is first registered with the register_netdev() function, then used in a number of functions that attach, turn on, and eventually unregister the device. LXDs provide a mechanism of *remote references* to refer to a specific object across domain boundaries. Similar to a capability in the LXD microkernel, each remote reference is a number that is resolved through a fast hash that is private to each thread of execution. References are transparent to the code, the IDL generates all necessary code to pair every local object with a reference that is used to lookup a corresponding shadow copy in another domain.

Function pointers Many parts of the kernel rely on the concept of an interface that allows dynamic registration of a specific subsystem implementation. In a native language like C an interface is implemented as a data structure with a collection of function pointers that are defined by each subsystem that provides a concrete interface implementation. In our example, net_device_ops is a data structure that defines a collection of function pointers implemented by the network device. We implement support for export of function pointers that cross boundaries of isolated domains. The following code provides a definition of the projection for the net_device_ops data structure.

```
projection <struct net_device_ops> net_device_ops {
  rpc [alloc] int (*ndo_open)(projection netdev_empty [bind] *dev);
  rpc [alloc] int (*ndo_stop)(projection netdev_empty [bind] *dev);
  rpc [alloc] int (*ndo_start_xmit)(projection sk_buff *skb,
                    projection net_device [bind] *dev);
  ...
}
```

For every function pointer, the IDL generates caller and callee stubs that behave like normal function pointers and hide details of cross-domain communication. To implement cross-domain function pointers while providing unmodified function signatures, we implement a concept of *hidden arguments*. For each function pointer, the IDL dynamically generates an executable trampoline in the caller's address space. The caller invokes this trampoline like any other function, however, the trampoline resolves additional hidden arguments as an offset from its own address. The hidden arguments describe which channel to use and passes this information to the cross-domain stub generated by the IDL compiler. A remote reference to a function pointer on the callee side allows the caller to resolve a specific instance of a function pointer.

Implementation We implement the IDL compiler as a source-to-source translator from the LXD IDL to C. To build the compiler, we rely on the formalism of parsing expression grammars (PEG). This choice allows us to design a modular grammar that is easy to extend with new IDL primitives. We use Vembyr PEG parser generator [63] to automate development of a compiler. Vembyr provides a convenient extension interface that allows us to construct an abstract syntax tree (AST) as a set of C++ classes. We then perform a compilation step as a series of passes over the AST, e.g., module import, derivation of directional attributes, etc.. The final pass converts the AST into a concrete syntax tree (CST) that we use to print out the C code.

```
1  DO_FINISH({
2    while (skb) {
3      struct sk_buff *next = skb->next;
4      ASYNC({
5        ...
6        rc = ndo_start_xmit(skb, dev);
7        ...
8      });
9      skb = next;
10   }
11 });
```

Listing 1: Asynchronous threads.

3.2 Asynchronous Execution Runtime

Traditionally, asynchronous communication requires explicit message passing [5, 42]. Programming of asynchronous message-passing systems, however, is challenging as it requires manual management (saving and restoring) of computation as execution gets blocked on remote invocations. In general, message-based systems work well as long as they are limited to a simple run-to-completion loop, but become nearly impossible to program if multiple blocking invocations are required on the message processing path [2, 52]. Further, in a message-passing environment, re-use of existing kernel code becomes hard or even impossible.

With LXDs we aim to satisfy two contradicting goals: 1) utilize asynchrony for cross-domain invocations, and 2) provide backward compatibility with existing kernel code, i.e., avoid re-implementation of the system in a message-passing style. To meet these goals, we implement a lightweight runtime that hides details of asynchronous communication behind an interface of asynchronous threads.

The core of the LXDs asynchronous runtime is built around two primitives: ASYNC() and DO_FINISH(). In Listing 1 the ASYNC() macro creates a new lightweight thread for executing a block of code (lines 4–8) asynchronously. Our implementation is based on GCC macros as it allows us to avoid modifications to the compiler and therefore provides compatibility with the existing kernel toolchain. When ndo_start_xmit() blocks on sending a message to the isolated driver (line 6), the asynchronous runtime continues execution from the next line after the asynchronous block (line 9) and starts the next iteration of the loop creating the new asynchronous thread.

Instead of blocking on the first ndo_start_xmit() cross-domain invocation, we dispatch multiple asynchronous invocations, hence submitting multiple network packets to the driver in a pipelined manner.

Internally, ASYNC() creates a minimal thread of execution by allocating a new stack and switching to it for execution of the code inside of the asynchronous block. ASYNC() creates a continuation, i.e., it saves the point of execution that follows the asynchronous block, which allows the runtime to resume execution when the thread either blocks or finishes. We save the state of the thread, i.e., its callee saved registers, on the stack, and therefore, can represent continuation as a tuple {instruction pointer, stack pointer}. The continuation is added to the run-queue that holds all asynchronous threads that are created in the context of the current kernel thread. When asynchronous thread blocks waiting on a reply from an isolated domain, it invokes the yield() function that again saves the state of the thread by creating another continuation that is added to the run-queue. The yield() function picks the next continuation from the run-queue and switches to it.

The DO_FINISH() macro specifies the scope in which all asynchronous threads must complete. When execution reaches the end of the DO_FINISH() block, the runtime checks if any of unfinished asynchronous threads are still on the run-queue. If yes, the runtime creates a continuation for the current thread knowing that it has to finish the DO_FINISH() block later, and switches to a thread from the run-queue.

Integration with the messaging system Every time a remote invocation blocks waiting on a reply, the asynchronous runtime switches to the new thread. The runtime system checks the reply message ring for incoming messages and whether any of them can unblock one of the blocked asynchronous threads. We implement a lightweight data structure that allows us to resolve response identifiers into pointers to asynchronous threads waiting on the run-queue. If the response channel is empty, the runtime system tries to return to the main thread to dispatch more asynchronous threads, but if the main thread reached the end of the DO_FINISH() block it picks one of the existing threads from the run-queue.

Nested invocations In most cases a cross-domain invocation triggers one or more nested remote invocations back into the caller domain, be it an LXD or a non-isolated kernel. For example, the ndo_start_xmit() triggers invocation of the consume_skb() function that releases the skb after it is sent. We need to process nested invocations in the caller domain. To avoid using an extra thread to dispatch remote invocations, we process nested invocations in the context of the caller thread. We embed a dispatch loop, the optimization we call "sender's dispatch loop", inside the message receive function thc_ipc_poll_recv() in such a way that it listens and processes incoming invocations from the callee.

Implementation ASYNC() and DO_FINISH() leverage functionality of GCC macros that allow us to declare the block of code as a nested function that can be executed on a new stack. We base implementation of the threading runtime on the eager version of AC [39] (in LXDs cross-domain invocations always block, therefore, eager creation of the stack for each asynchronous thread is justified). Besides changing AC to work inside the Linux kernel and integrating it with the LXDs messaging primitives, we employ several aggressive optimizations. To minimize the number of thread switches, we introduce an idea of a "direct" continuation, the continuation that is known to follow the current context of execution, e.g., the instruction following the ASYNC() block. We also defer deallocation of stacks. Normally, the stack cannot be deallocated from the context of the thread that is using it. AC switches into the context of the "idle" scheduler thread and deallocates the stack from there. This, however, introduces an extra context switch. Instead, we maintain the queue of stacks pending de-allocation and deallocate them all at once right when the execution exits the DO_FINISH() block.

3.3 Fast Cross-Core Messaging

Trying to reduce overheads of crossing the isolation boundary, LXDs schedule isolated subsystems on separate CPU cores and use a fast cross-core communication mechanism to send "call" and "reply" invocations between the cores. The performance of cross-core invocations is dominated by the latency of cache coherence protocol, which synchronizes cache lines between cores (a single cache-line transaction incurs a latency of 100-400 cycles [22, 56, 57]). In order to achieve the lowest possible communication overhead, similar to prior projects [5,6,47], we minimize the number of cache coherence transactions. In LXDs, each channel consists of two rings: one for outgoing "call" messages and one for incoming "replies". We configure each message to be the size of a single cache line (64 bytes on our hardware). Similar to FastForward [36], we avoid shared producer and consumer pointers, as they add extra transactions for each message. Instead, we utilize an explicit state flag that signals whether the ring slot is free.

4 Decomposition Case-Studies

To evaluate the generality of LXDs abstractions, we develop several isolated device drivers in the Linux kernel.

4.1 Network Device Drivers

We develop isolated versions of two network drivers: 1) a software-only dummy network driver that emulates an infinitely fast network adapter, and 2) Intel 82599 10Gbps Ethernet driver (ixgbe). The network layer of the Linux kernel has one of the tightest performance budgets among all kernel subsystems. Further, the dummy is not connected to a real network interface, and hence allows us to stress overheads of isolation without any artificial limits of existing NICs.

Decomposed network drivers To isolate the dummy and ixgbe network drivers, we develop IDL specifications of the network driver interface. The IDL specification is 64 lines of

code for the dummy, and 153 lines for the ixgbe driver (110 lines for the network and 43 for the PCIe bus interfaces). Each network device driver registers with the kernel by invoking a kernel function and passing a collection of function pointers that implement an interface of a specific driver. Since ixgbe manages a real PCIe device, it registers with the PCIe bus driver that enumerates all PCIe devices on the bus and connects them to matching device drivers. Therefore, we develop an IDL specification for the PCI bus interface.

In contrast to block device drivers that implement a zero-copy path for block requests, the network stack copies each packet from a user process into a freshly allocated kernel buffer. To ensure a zero-copy transfer of the packet from the kernel to the LXD, we allocate a region of memory shared between the non-isolated kernel and the LXD of the network driver. The kernel allocates memory for the skb payload using the alloc_skb() function. We modify it to allocate payload data from this shared region. Linux does not provide a simple mechanism to configure one of its memory allocators to run on a specified region of memory. We, therefore, develop a lock-free allocator that uses a dedicated memory region to allocate blocks of a fixed size. To enable device access to the region of shared memory where packet payload is allocated, we extend the libLXD and the LXD microkernel with support for the IOMMU interface. We configure the IOMMU to enable access to the packet payload region that is shared between the kernel and the LXD.

The ixgbe device driver uses system timers for several control plane operations. To provide timers inside LXDs, we rely on the timer infrastructure of the non-isolated kernel. Much like any other function pointer, we register the timer callback function pointer with the kernel. The callback caller stub sends an IPC to the isolated driver to trigger the actual callback inside the LXD. Finally, in the native driver, the NAPI polling function is invoked in the context of the softirq thread. We implement softirq threads as asynchronous threads dispatched from the LXD's dispatch loop.

The isolated dummy driver requires two cross-domain calls on the packet transmission path. The first call is invoked by the non-isolated kernel to submit the packet to the driver (ndo_start_xmit()), and the second is called by the driver after the packet was processed by the device and is ready to be released (consume_skb()). To reduce overheads of isolation, we introduce the "half-crossing" optimization. Specifically, for the functions that do not return a value, e.g., consume_skb() that releases the network packet to the kernel, we send the "call" message across the isolation boundary, but do not wait for the arrival of the reply message.

4.2 Multi-Queue Block Device Drivers

We implement a decomposed version of the nullblk block driver [4]. The nullblk driver is not connected to a real NVMe device, but instead emulates the behavior of the fastest possible block device in software.

Linux multi-queue block layer Linux implements a multi-queue (MQ) block layer [7] to support low-latency, high-throughput PCIe-attached non-volatile memory (NVMe) block devices. On par with network adapters, today NVMe is one of the fastest I/O subsystems in the kernel. To fully benefit from the asynchronous multi-queue layer, user-level processes rely on the new asynchronous block I/O interface that allows applications to submit batches of I/O requests to the kernel and poll for completion later. In the case of direct device access, the kernel performs all request processing starting from the system call to leaving the request ready for the DMA in the context of the same process that issued the io_submit() system call. The kernel returns to the process right after leaving request in the DMA ring buffers. Later the process polls for completion of the request by either entering the kernel again, or by monitoring a user-mapped page where the kernel advertises completed requests. Being allocated inside a user-level page, the pointer to the request is passed to the kernel, the kernel "pins" the page ensuring that it does not get swapped out while the request is in-flight. For each request the device driver adds the page containing the request to the IOMMU of the device, hence permitting the direct access to the request payload.

Decomposed block driver Similar to network drivers, we develop IDL specifications of the block driver interface, which consists of 68 lines of IDL code. The isolated nullblk driver requires three cross-domain calls on the I/O path. The first call is invoked by the non-isolated kernel to pass a block request from the block layer to the driver. The driver itself invokes two functions of the non-isolated kernel: blk_mq_start_request() and blk_mq_end_request(). The blk_mq_start_request() function passes the pointer to the request back to the block layer to inform it that request processing has started, and the I/O is ready to be issued to the device. The block layer now associates a timer with this particular request to ensure that if the completion for that request does not arrive in time, it can either abort the I/O operation or try to enqueue the request again. The blk_mq_end_request() allows the driver to inform the block layer that the request is completed by the device, and is ready to go up the block layer back to the user process.

We utilize ASYNC() and DO_FINISH() to implement an asynchronous loop on the submission path. A batch of requests is submitted by the application, hence we dispatch them to the nullblk LXD asynchronously. We provide a detailed analysis of isolation overheads in Section 5.6.

5 Evaluation

We conduct all experiments in the openly-available CloudLab network testbed [65].[3] We utilize CloudLab d820 servers with four 2.2 GHz 8-Core E5-4620 processors and 128 GB RAM. All machines run 64-bit Ubuntu 18.04 Linux with the kernel version 4.8.4. In all experiments we disable hyper-threading,

[3] LXDs are available at https://github.com/mars-research/lxds.

Operation	Cycles (Cycles per request)
Context switch	29-41
1 non-blocking ASYNC()	46
1 blocking ASYNC()	124
4 blocking ASYNC()s	374 (93.5)

Table 1: Overhead of asynchronous threads.

Operation	Cycles
seL4 same-core d820 (without PCIDs)	1005
seL4 same-core c220g2 (with PCIDs)	834
LXDs cross-core r320 (non-NUMA)	448
LXDs cross-core d820 (NUMA)	533

Table 2: Intra-core vs cross-core IPC.

turbo boost, and frequency scaling to reduce the variance in benchmarking.

5.1 Asynchronous Runtime

LXDs rely on the asynchronous runtime to hide the overheads of cross-domain invocations. To evaluate the effectiveness of this design choice, we conduct two sets of experiments that measure and compare overheads of asynchronous threads, and synchronous invocations.

Overhead of asynchronous threads We conduct four experiments that measure overheads of the asynchronous runtime (Table 1). In all tests we run 10M iterations and report an average across five runs. The first test measures the overhead of creating and tearing down a minimal asynchronous block of code that just increments an integer, but does not block. Each iteration takes 46 cycles which includes allocating and deallocating a stack for the new thread, and two stack switches to start and end execution of the thread. In the second test we measure the overhead of switching between a pair of asynchronous threads that takes 29 cycles and uses a sequence of 20 CPU instructions. Out of 20 instructions 16 are memory accesses that touch the first level cache and take two cycles each [28] (six instructions are required to save and restore callee saved registers, and two save and restore instruction pointer and stack registers). If, however, the context switch touches additional metadata, e.g., adds the thread to the runqueue, the overhead of the context switch grows to 41 cycles due to additional memory accesses.

The third and fourth tests measure the overhead of executing one and four blocking ASYNC() code blocks, i.e., each thread executes yield() similar to the IPC path. The overhead of creating one blocking asynchronous block (third test in Table 1) is 124 cycles, which consist of the cost to create and tear down a non-blocking asynchronous thread (46 cycles) and three context switches required to block and unblock the thread, and switch back to the main thread when the DO_FINISH() block is reached. If, however, we execute four asynchronous blocks in a loop the total overhead comes to 374 cycles or 93.5 cycles per one asynchronous block. Overall, we conclude that asynchronous threads are fast, and come close to the speed of manual management of pending invocations in a message-passing system.

Batch size	Cycles (cycles per msg)	
	Manual	ASYNC()
1	533	568
4	876 (219)	1111 (277)
8	1262 (157)	2096 (262)

Table 3: Benefits of manual and ASYNC() batching.

5.2 Same-core vs cross-core IPC

Same-core IPC To understand the benefits of cache-coherent cross-core invocations over traditional same-core address-space switches, we compare LXDs' cross-core channels with the synchronous IPC mechanism implemented by the seL4 microkernel [26]. We choose seL4 as it implements the fastest synchronous IPC across several modern microkernels [55]. As d820 servers do not provide support for tagged TLBs (PCIDs) that improve IPC performance by avoiding an expensive TLB flush on the IPC path, in addition to the d820 machines we report results for the same IPC tests on an Intel E5-2660 v3 10-core Haswell 2.6GHz machine (CloudLab c220g2 server) that implements support for tagged TLBs. To defend against Meltdown attacks, seL4 provides support for a page-table-based kernel isolation mechanism similar to KPTI [37]. However, this mechanism negatively affects IPC performance due to an additional reload of the page table root pointer. Since recent Intel CPUs address Meltdown attacks in hardware, we configure seL4 without these mitigations. On d820 machines without PCIDs support, seL4 achieves the median IPC latency of 1005 cycles (Table 2). On the c220g2 servers with tagged TLBs enabled the IPC latency drops to 834 cycles (Table 2).

Cross-core IPC To measure the overhead of cross-core cache-coherent invocations, we conduct a minimal call/reply test in which a client thread repeatedly invokes a function of a server via an LXD's asynchronous communication channel. Client and server are running on two cores of the same CPU socket. Since multi-socket NUMA machines incur higher cache-coherence overheads and thus have slower cross-domain invocations, in our experiments we a NUMA and a non-NUMA machine with a similar CPU: a four socket d820 NUMA server and a single-socket non-NUMA r320 CloudLab server configured with one 2.1 GHz 8-core Xeon E5-2450 CPU. In all experiments we run 100M call/reply invocations and report an average across five runs (Table 2). On a non-NUMA r320 machine, cross-core IPC takes 448 cycles. On a NUMA d820 machine, this number increases to 533 cycles.

Two additional observations are important. First, communication between hardware threads of the same CPU core takes less time than communication between cores (we measure the overhead of cross-core invocations to be only 105 cycles on the non-NUMA r320 machine and 133 cycles on the NUMA

d820). Typically, however, a single LXD serves requests from multiple cores of the monolithic kernel, and hence only a single core can benefit from proximity to the logical core.

Second, communication outside of the NUMA node incurs high overheads due to crossing inter-socket links. On the d820 server, a cross-socket call/reply invocation takes 1988 cycles over one inter-socket hop, which is higher than overhead of a synchronous same-core IPC. Note that on a batch of 4 and 8 this number drops to 900 and 535 cycles per message respectively. We anticipate that each NUMA node will run a local LXD thread and hence the crossings of NUMA nodes will be rare (this design makes sense as high-throughput isolated subsystems, e.g., network and NVMe drivers, are CPU-bound and anyway require multiple LXD threads to keep up with invocations from multiple kernel threads).

Finally, we make an observation that compared to overheads of synchronous IPC invocations (both on the same core and cross-core) the overheads of asynchronous threads is relatively small (93.5 cycles per-request in a batch of four (Table 1)). Therefore, the use of asynchronous threads for batching and pipelining of multiple cross-domain invocations is justified.

5.3 Message Batching

To evaluate the benefits of aggregating multiple cross-core invocations in a batch, we conduct an experiment that performs call/reply invocations in batches of messages ranging from 1 to 8. On a batch of 4 messages a call/reply invocation takes only 876 cycles, or 219 cycles per invocation on a d820 NUMA machine (Table 3). On a batch of 8 messages the overhead per one call/reply invocation drops to 157 cycles per message. For a batch of messages, the cross-core IPC sends call/reply invocations through independent cache lines. The CPU starts sending the next message right after issuing loads and stores to the hardware load/store queue, but without waiting for completion of the cache-coherence requests effectively pipelining multiple outstanding cache coherence requests.

Composable batching with ASYNC() Finally, we analyze how cross-core invocations are affected if the batches of messages are created by blocking asynchronous threads instead of the manual, message-passing style batching we analyzed above. To evaluate overheads of asynchronous threads, we design an IPC test that performs a series of cross-core function invocations from inside an ASYNC() block (Table 3). We run a loop of length 1, 4, and 8. The body of the loop is an asynchronous code block that invokes a function on another core. Instead of waiting for the reply, each asynchronous thread yields and continues to the next iteration of the loop that dispatches the new asynchronous thread. For the loop of length 1, 4, and 8, compared to the manual batch, ASYNC() introduces overhead ranging from 35 cycles on a batch of one to 105 cycles per message on a batch of 8 (Table 3).

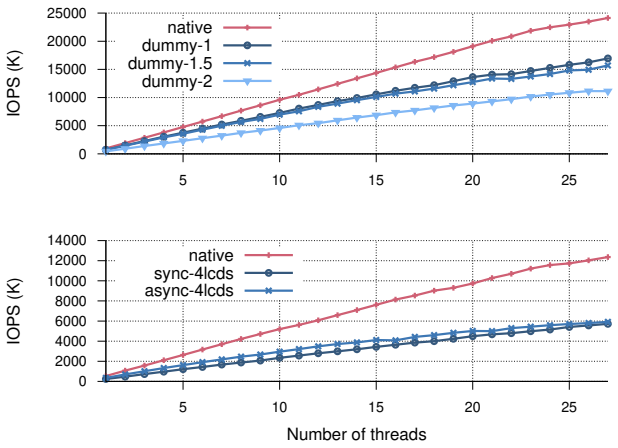

Figure 3: Performance of the dummy driver

5.4 Dummy Device Driver

We utilize the dummy driver as a platform for several benchmarks that highlight overheads of isolation in the context of a "fast" device driver (dummy is a good, representative example of such device driver as it serves an infinitely fast device and is accessed through a well-optimized I/O submission path of the kernel network stack). In all experiments we use the iperf2 benchmark that measures the transmit and receive bandwidth for different payload sizes, and run the tests on d820 servers (Figure 3). We configure the isolated dummy driver with a varying number of cores ranging from one to four in such a manner that one LXD thread runs on each socket of the 4-socket d820 system. Specifically, on a 32-core system, the isolated dummy can support up to 27 iperf threads (i.e., four cores of the system are dedicated to LXD threads, and one core is occupied by the kLXD thread servicing control plane invocations from the LXD). We assign the first six iperf threads to the first socket (one core of the CPU socket is occupied by the LXD thread and one by the kLXD thread), then we assign the next seven iperf threads (7-13) to the next socket, and so on (Figure 3). We report the total number of device driver I/O requests per-second (IOPS) across all threads (we report an average across five runs on the maximum transmission unit (MTU) size packets).

In our first experiment we change dummy to perform only *one* crossing between the kernel and the driver for sending each packet (`dummy-1`, Figure 3). This synthetic configuration allows us to analyze overheads of isolation in the ideal scenario of a device driver that requires only one crossing on the device I/O path. With one application thread the non-isolated driver achieves 956K IOPS (i.e., on average, a well-optimized network send path takes only 2299 cycles to submit an MTU-sized packet from the user process to the network interface). The isolated driver achieves 730K IOPS (76% of the non-isolated performance), and on average requires 3009 cycles to submit one packet. Of course, the isolated driver utilizes one extra core for running the LXD. Isolation adds an overhead of

710 cycles per-packet, which includes the overhead of the IPC and processing of the packet by the driver (in this experiment, LXDs do not benefit from any asynchrony; all packets are submitted synchronously). On 27 threads the isolated driver achieves 70% of the performance of the non-isolated driver. Compared to the configuration with one application thread, the slight drop in performance is due to the fact that each of four LXDs service up to seven application threads which adds overhead to the LXD's dispatch processing loop.

In practice, the dummy driver requires two domain crossings for submitting each packet (Section 4). We evaluate how performance of isolated drivers degrades with the number of crossings by running a version of dummy that performs two full cross-domain invocations (`dummy-2`, Figure 3). On one thread, two crossings add overhead of 1794 cycles per packet. The "half-crossing" optimization, however, reduces the overhead of two crossings from 1794 cycles per-packet to only 814 cycles (`dummy-1.5`, Figure 3).

Asynchronous threads To evaluate the impact of asynchronous communication, we perform the same iperf2 test with a packet size of 4096 bytes. When the packet size exceeds MTU, the kernel fragments each packet into MTU-size chunks suitable for transmission and submits each chunk to the driver individually. In general, multiple domain crossings caused by fragmentation negatively affect performance of the isolated driver. We compare three configurations: a non-isolated dummy driver (`native`, Figure 3), a synchronous version of LXDs (`sync-4-lcds`) and asynchronous version that leverages ASYNC() to invoke the driver in a parallel loop (`async-4-lcds`). Configured with one iperf thread, a non-isolated driver achieves 534K IOPS, i.e., on average it requires 4114 cycles to submit a 4096 byte packet split in three fragments. Performance of the synchronous version of the isolated driver is heavily penalized by the inability to overlap communication, i.e., waiting for LXD replies, and processing of further requests. The synchronous version achieves only 236K IOPS (44% of non-isolated performance). The asynchronous isolated driver is able to benefit from pipelining of three fragmented packets with asynchronous threads (it achieves 341K IOPS or 63.8% of non-isolated performance). Note, that as the number of application threads grows, the benefits of asynchronous threads gradually disappear. With 27 iperf threads both synchronous and asynchronous configurations achieve similar performance (36% and 37% of the native driver respectively). As the number of application threads increases, each core of the isolated driver that processes requests from up to seven iperf threads becomes heavily utilized. Each LXD thread dispatches kernel invocations in a round-robin manner from a set of cross-core communication channels. If all channels are active, the performance of each iperf thread is dominated by the time spent waiting for its turn to be processed by the LXD. On a batch of only three messages, asynchronous threads do not provide sufficient benefits to tolerate this latency.

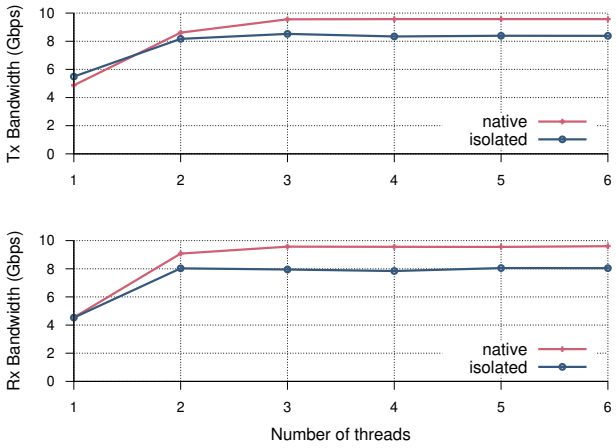

Figure 4: Ixgbe Tx and Rx bandwidth.

5.5 Ixgbe Device Driver

To measure performance of the isolated ixgbe driver, we configure an iperf2 test with a varying number of iperf threads ranging from one to six (Figure 4). On our system, a small number of application threads saturates a 10Gbps network adapter. Configured with one iperf thread, on the MTU size packet the isolated ixgbe is 12% faster compared to the isolated system on the network transmit path, although at the cost of using an extra core. This advantage disappears as the LXD becomes busy handling more than one iperf thread. Nevertheless, from three to seven threads, the isolated driver stays within 6-13% of the performance of the native device driver which saturates the network interface with three and more application threads.

On the receive path, the isolated driver is 1% slower for one application thread. Two factors attribute to performance of the isolated driver: 1) it benefits from an additional core, and 2) it uses asynchronous threads for NAPI polling instead of native threads used by the Linux kernel for handling IRQs. Asynchronous threads provide a faster context switch compared to the native Linux kernel threads. Similar to transmit path, this advantage disappears with larger number of application threads. From two to six threads the isolated driver stays within 12-18% of the performance of the native driver.

To measure the end-to-end latency, we rely on the UDP request-response test implemented by the netperf benchmarking tool. The `UDP_RR` measures the number of round-trip request-response transactions per second, i.e., the client sends a 64 byte UDP packet and waits for the response from the server. The native driver achieves 26688 transactions per second (which equals the round-trip latency of 40μs), the isolated driver is 7% (2.6μs) faster with 24975 transactions per second (round-trip latency of 37.4μs). Again the isolated driver benefits from a faster receive path due to low-overhead context switch of asynchronous threads. As the network is lightly loaded during the latency test even with six application threads the isolated driver remains 3.4μs faster achieving the latency

of 43.4μs versus 46.8μs achieved by the native driver.

5.6 Multi-Queue Block Device Driver

In our block device experiments, we use fio to generate I/O requests. To set an optimal baseline for our evaluation, we chose the configuration parameters that provide the lowest latency path to the driver, so that overheads of isolation are emphasized the most. We use fio's libaio engine to overlap I/O submissions, and bypass the page cache by setting direct I/O flag to ensure raw device performance. Similar to dummy, in isolated configuration, the nullblk LXD fully utilizes one extra core on every CPU socket. We run the same configurations as for dummy, e.g., one LXD thread on each NUMA node. We placing the first six fio threads on the first NUMA node, next seven fio threads on the second NUMA node, and so on, up until 27 fio threads. We vary the number of fio threads from 1 to 27 and report results for two block sizes—512 bytes and 1MB—which represent two extreme points: a very small and a very large data block. For each block size, we submit a set of requests at once ranging the number of requests from 1 to 16 and then poll for the same number of completions. Since the nullblk driver does not interact with an actual storage medium writes perform as fast as reads, hence we utilize read I/O operations in all experiments.

The native driver achieves 295K IOPS for the packet size of 512 bytes and the queue of one (Figure 5). In other words, a single request takes about 7457 cycles to complete. The isolated driver achieves 235K IOPS (or 79% of non-isolated performance). The isolation incurs an overhead of 1904 cycles due to three domain crossings on the critical path. For a queue of 16 requests, the isolated driver benefits from asynchronous threads which allow it to stay within 4% of the performance of the native driver for as long as it stays in one NUMA node (from 1 to 6 fio threads). Both native and isolated drivers suffer from NUMA effects due to the fact that Linux block layer collects performance statistics for every device partition. The blk_mq_end_request() function acquires a per-partition lock and updates several global counters. The native driver faces performance drops when it spills outside of a NUMA node at 9, 17, and 25 fio threads (Figure 5). The isolated driver experiences similar drops at 7, 14, and 21 fio threads. On the block size of 1M, inside one NUMA node the isolated driver stays within 10% of the performance of the native driver for both queues of one and 16 requests. Outside of one NUMA node the performance of both native and isolated drivers suffers from NUMA effects. We speculate that NUMA degradation can be fixed by changing the kernel to use per-core performance counters [10].

6 Conclusions

LXDs provide general abstractions and mechanisms for isolating device drivers in a full-featured operating system kernel. By employing several design choices—relying on an asynchronous execution runtime for hiding latency of cross-domain invocations, developing general decomposition patterns, and relying on cross-core invocations—we demonstrate the ability to isolate kernel subsystems with tightest performance budgets. We hope that our work will gradually enable kernels to employ practical isolation of most device drivers and other kernel subsystems that today account for the majority of the kernel code.

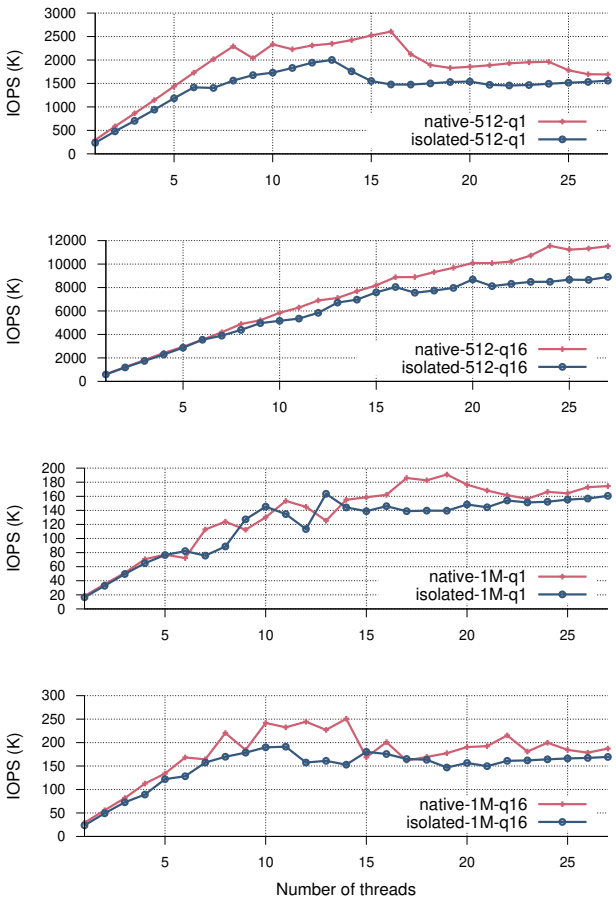

Figure 5: Performance of the nullblk driver

Acknowledgments

We thank ASPLOS 2018, OSDI 2018, and USENIX ATC 2019 reviewers and our shepherd, Andrew Baumann, for in-depth feedback on earlier versions of the paper and numerous insights. Also we would like to thank the Utah Emulab and CloudLab team, and especially Mike Hibler, for his continuous support and endless patience in accommodating our hardware requests. This research is supported in part by the National Science Foundation under Grant Numbers 1319076, 1527526, and 1817120 and Google.

References

[1] Code-Pointer Integrity in Clang/LLVM. https://github.com/cpi-llvm/compiler-rt.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J.

Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference (ATC)*, pages 289–302, Berkeley, CA, USA, 2002.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139(140):116, 2005.

[4] Jens Axboe. Null block device driver. https://www.kernel.org/doc/Documentation/block/null_blk.txt, 2019.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44, New York, NY, USA, 2009.

[6] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, 1991.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 22:1–22:10, New York, NY, USA, 2013.

[8] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 30–40, 2011.

[9] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Berkeley, CA, USA, 2010.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX Annual Technical Conference (ATC)*, pages 9–22, 2010.

[12] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[14] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[15] Citrix. XenClient. http://www.citrix.com/products/xenclient/how-it-works.html.

[16] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 189–202. ACM, 2011.

[17] Microsoft Corporation and Digital Equipment Corporation. The component object model specification, 1995.

[18] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, and Jonathan Walpole. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[19] CVE Details. Vulnerabilities in the Linux kernel by year. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[20] CVE Details. Vulnerabilities in the Linux kernel in 2018. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/Linux-Linux-Kernel.html.

[21] Data61 Trustworthy Systems. *seL4 Reference Manual*, 2017. http://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

[22] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 33–48. ACM, 2013.

[23] Distributed component object model (DCOM) remote protocol specification. https://msdn.microsoft.com/library/cc201989.aspx.

[24] DDEKit and DDE for Linux. http://os.inf.tu-dresden.de/ddekit/.

[25] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *ACM SIGPLAN Notices*, volume 32, pages 44–56. ACM, 1997.

[26] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 133–150. ACM, 2013.

[27] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture*. Technische Universität, Dresden, Fakultät Informatik, 2007.

[28] Agner Fog. Instruction tables. http://www.agner.org/optimize/instruction_tables.pdf.

[29] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 38–51, 1997.

[30] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[31] Linux FUSE (filesystem in userspace). https://github.com/libfuse/libfuse.

[32] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[34] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114. ACM, 2000.

[35] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 2002.

[36] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2008.

[37] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[38] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, October 22 2000.

[39] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. AC: composable asynchronous IO for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

[40] Härtig, H. Security architectures revisited. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 16–23. ACM, 2002.

[41] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[42] Herder, J.N. and Bos, H. and Gras, B. and Homburg, P. and Tanenbaum, A.S. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[43] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[44] Nick Holloway. Dummy net driver. https://elixir.bootlin.com/linux/latest/source/drivers/net/dummy.c, 1994.

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

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

[47] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–48, 2016.

[48] Antti Kantee. *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. PhD thesis, 2012.

[49] Vinay Katoch. Whitepaper on bypassing ASLR/DEP. http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf, 2011.

[50] Kenton Varda. Cap'n Proto Cerealization Protocol. http://kentonv.github.io/capnproto/.

[51] Kil3r and Bulba. Bypassing StackGuard and StackShield. *Phrack Magazine*, 53, 2000.

[52] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *USENIX Annual Technical Conference (ATC)*, pages 7:1–7:14, 2007.

[53] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[54] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *Workshop on I/O Virtualization*, 2011.

[55] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.

[56] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[57] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270. IEEE, 2009.

[58] Tilo Müller. ASLR smack and laugh reference. *Seminar on Advanced Exploitation Techniques*, 2008.

[59] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, 2001.

[60] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 116–132, New York, NY, USA, 2013.

[61] Object Management Group. OMG IDL Syntax and Semantics. http://www.omg.org/orbrev/drafts/3_idlsyn.pdf.

[62] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[63] Jon Rafkind. Vembyr - multi-language PEG parser generator written in Python, November 2011. http://code.google.com/p/vembyr/.

[64] Matthew J Renzelmann and Michael M Swift. Decaf: Moving device drivers to a modern language. In *USENIX Annual Technical Conference (ATC)*, 2009.

[65] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:*, 39(6):36–38, December 2014.

[66] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[67] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–8, 2010.

[68] Green Hills Software. INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[69] 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. ACM, 2002.

[70] Hajime Tazaki. An introduction of library operating system for Linux (LibOS). https://lwn.net/Articles/637658/.

[71] Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59, 2002.

[72] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.8, update 3. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[73] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog*, 2008.

[74] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[75] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondrian memory protection. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 31–44. ACM, 2005.

[76] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[77] 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 *Usenix Security Symposium*, 2018.