Abstract

Even after decades of work to make monolithic kernels more secure, serious vulnerabilities in them are still reported every year. Because the entire monolithic kernel is in one address space, an attacker is just one vulnerability away from owning the entire machine. We argue that it is time to decompose monolithic kernels like Linux into smaller parts that run in isolated compartments and communicate using secure interfaces. We think this is timely due to recent trends in hardware that make it easier and efficient to isolate kernel components.

In this work, we describe our initial steps toward this goal. We implemented a small microkernel module that is installed in Linux and manages Lightweight Capability Domains (LCDs) that are implemented using hardware virtual machines. We describe our implementation and the challenges we encountered in attempting to run unmodified kernel code in isolation.

1. Project Vision

Modern, monolithic kernels are a big security liability for the systems they run on. They consist of complex subsystems that interoperate with no isolation, while fulfilling security-critical roles like sandboxing processes and mediating access to insecure interfaces. One small bug in a kernel driver or subsystem can be exploited by an attacker to take down the entire machine. To prevent these vulnerabilities from arising, researchers and developers have come up with a variety of solutions, including NX stacks, address space layout randomization, formal verification techniques, testing, and static analysis techniques [5, 9]. Unfortunately, kernel vulnerabilities are still found regularly, and the number reported each year is staying steady or increasing: The Common Vulnerabilities and Exposures database lists hundreds of vulnerabilities for kernels like Linux that allow attackers to do denial-of-service, privilege escalation, and code injection attacks [1].

Why do we use monolithic kernels? There are a number of reasons. First, monolithic kernels have reached critical mass with large scale developer support, and they provide a unified interface for various platforms. Second, until recently, system architectures have largely remained homogeneous with a small number of cores inside a single cache coherence domain and unified address space. On such architectures, isolated kernel subsystems would end up sharing resources like CPUs, leading to expensive context switching and multiplexing while handling a single system call. Third, there has been little motivation to transition to a distributed kernel, despite recent hardware trends. This is because building such a distributed kernel from scratch that provides support for all platforms would take years, and decomposing a monolithic kernel into isolated subsystems would also take a lot of effort: Subsystems inside a monolithic kernel interact in complex and poorly documented ways and rely on shared memory.

But recently, monolithic kernels are being pushed out of the data plane in some application stacks as programmers take advantage of new advances in hardware. For example, the Arrakis operating system [12] uses hardware virtualization to isolate applications and allows them to run ‘bare metal’ with direct access to hardware. Arrakis runs Linux on a separate set of processor cores to handle control plane operations like managing a file system. We think systems like Arrakis are part of a trend that will continue as systems designers try to keep up with advances in hardware, like increasing core counts, dark silicon, and heterogeneous cores on a single chip. Unfortunately, applications designed for systems like Arrakis run as isolated ecosystems, and it’s difficult to compose such applications.

We argue that it is time to decompose a monolithic kernel like Linux in order to make it more secure and to address composability issues in systems like Arrakis. As in Barreelfish [2], we envision running code in isolated domains that communicate using secure, capability-mediated message passing and shared memory. But some domains will contain decomposed kernel components that can be used to link application stacks into powerful pipelines. For example, an NFS server could be linked with an optimized network stack and storage stack using decomposed TCP/IP and block layer components from the Linux kernel, as shown in Figure 1.
But since Linux will be difficult to decompose, we will provide a few key features to make it easier. First, we will install a microkernel inside Linux that manages the isolated domains that run inside hardware virtual machines, similar to KVM. Non-isolated code will run without modification, and the isolated domains will be scheduled by the Linux scheduler. This helps us achieve the strong isolation we need but without needing to decompose the entire kernel in one phase. Second, we will provide an interface definition language (IDL) that is powerful enough to describe the interfaces between subsystems in Linux, which are often stateful and use shared data structures. We will also provide a compiler that translates an IDL specification into low-level message passing code that can be linked with unmodified kernel code and resolve dependencies. We think this is feasible because of all the effort to design the object-oriented interfaces that are currently in the kernel.

2. My Work
As a step toward fulfilling our vision, my objectives this past year were to design and build (1) the execution environment and microkernel interface for isolated code, (2) the interface for non-isolated code, part of the microkernel internals, and (3) a proof-of-concept example that decomposes and isolates a file system. I succeeded in finishing most of the execution environment and interfaces, but I ran out of time for decomposing and isolating a file system. I will discuss the different parts of the microkernel module and the execution environments in more detail in the following sections, beginning with a conceptual overview.

3. Conceptual Overview
The code to be isolated is installed inside a Lightweight Capability Domain (LCD). Each LCD runs in its own hardware virtual machine and address space, and communicates with other LCDs and non-isolated code using message passing. We install a microkernel module in the non-isolated part of Linux that manages these VMs and provides a capability-mediated interface for LCDs to perform message passing (IPC), map pages in their address space, and so on. The microkernel module also provides a capability-mediated interface for non-isolated code to set up and interact with LCDs. See Figure 2.

Capabilities We followed the design of other microkernels, including seL4, for building the capability and IPC code in our microkernel. An LCD refers to objects managed by the microkernel, like IPC channels and physical pages, using a file-descriptor-like integer identifier called a capability pointer, or cptr. LCDs use cptrs as arguments to system calls when they call out into the microkernel. The microkernel uses an LCD’s capability space, or cspace, to resolve cptrs to capabilities. A capability contains a reference to a microkernel object along with the LCD’s access rights. The microkernel processes the system call only if the LCD’s rights allow it. We currently support ‘all or nothing’ access rights. Capability lookup is designed to be fast since it is a common operation; seL4 uses a sparse, radix-tree-like data structure called guarded page tables, and we used a similar design.

Synchronous IPC LCDs use synchronous IPC to send messages with scalar values and to grant capabilities to other LCDs. A sending LCD stores scalar
values in message registers and cptrs to capabilities it would like to grant in capability registers. It then invokes a send operation on an IPC channel, and blocks until another LCD receives the message on the channel.

Meanwhile, a receiving LCD stores cptrs to destination slots in its cspace where it would like granted capabilities to be stored into its capability registers, invokes a receive on an IPC channel, and blocks until another LCD sends a message on the channel. When the microkernel matches a sender with a receiver, it inspects the sender’s capability registers and copies the capability data from the sender’s cspace to the receiver’s cspace. The microkernel records a parent-child relationship between the sender’s and the receiver’s access rights for an object, so that a sender can fully revoke all rights it granted to the receiver, the receiver granted to another LCD, and so on.

**Execution Environments** Finally, we also need to provide two execution environments: One to support the unmodified kernel code inside an LCD, and another for the non-isolated kernel code so that it can interact with LCDs. The isolated code execution environment is more complicated since isolated code needs support for managing its address space and memory, invoking system calls to communicate with the microkernel, and so on.

4. Microkernel Interface & Thread Execution Environment

Non-isolated threads running in the host Linux kernel and LCDs use the same interface and high-level execution environment to do IPC, create other LCDs, allocate memory, and so on. LCDs are currently single-threaded. Before using any other part of the microkernel interface, a thread must execute `lcd_enter` to initialize the execution environment (similar to `cap_enter` in Capsicum). Each thread is provided with a set of message registers and capability registers that it can use during IPC and system calls, and simple routines to access them. A thread should store values in these registers before invoking a system call. Each thread is also associated with its own cspace that it can use, indirectly, to store references and access rights to microkernel objects. Finally, the microkernel does not choose a free slot in a thread’s cspace to place a capability; the thread must track which slots are free and provide a cptr to a free slot when it needs to store a capability in its cspace. Accordingly, each thread is provided with a ‘cptr cache’ to track which slots are available in its cspace.

An example in Figure 3 highlights the main features (return value checking and other details have been omitted). A thread executing this code creates an LCD to isolate the btrfs file system and an IPC endpoint to communicate with it. It grants access rights to the endpoint to the btrfs LCD, boots it, and begins a sequence of IPC messages. When the thread is finished, it deletes its capabilities to the endpoint and the LCD. So long as it did not grant any other thread rights to the LCD or endpoint, these objects will be destroyed. Note that the code does not show allocation of cptrs from the thread’s cptr cache because the interface routines do this on behalf of the thread, since it is a common operation. Note too that most of this code can run in an isolated or non-isolated environment (we have not implemented the entire interface for the isolated environment, like creating LCDs).

5. Non-Isolated Threads

Any thread running inside the non-isolated kernel can enter into ‘LCD mode’ by executing `lcd_enter` and begin using the microkernel interface. We call such threads ‘kernel LCDs’ or kLCDs since they use the same interface as isolated code and since the microkernel-related parts of their environment are similar to the isolated environment. We want kLCDs to use the same interface as isolated code so that all interactions between non-isolated and isolated code are mediated by the microkernel, and so that kLCD code can be isolated in the future if needed, with few changes. The non-isolated implementation of the microkernel interface, kLIBLCD, is logically separate from the microkernel, but is compiled and linked with the microkernel module and runs in the non-isolated address space. When a non-isolated thread invokes `lcd_enter`, kLIBLCD initializes a cptr cache, cspace, and message registers and attaches them to the thread. The thread should invoke `lcd_exit` before it terminates and is reaped by the host scheduler.

We designed kLIBLCD to use the Linux kernel’s module loading code for loading a kernel module inside an LCD. We made a small number of modifications to the module loading code so that if a kernel module is destined for an LCD, we don’t run its initialization routine until it is loaded in the LCD. We modified modprobe to use an ioctl interface we provide for loading a

---

Since threads in our design do not have as much control over building cspaces as in seL4, it may make more sense to have the microkernel choose free slots on the thread’s behalf.
init(cptr_t module_loader) {
    cptr_t e, b;
    /*
     * Initialize environment
     */
    lcd_enter();
    /*
     * Set up endpoint and LCD
     */
    e = lcd_create_endpoint();
    b = lcd_create_module_lcd("btrfs", module_loader);
    lcd_cap_grant(b, e);
    lcd_run(b);
    /*
     * Send and receive messages
     */
    lcd_set_r0(0xbadf00d);
    lcd_set_r1(0x1234567);
    lcd_send(e);
    ...
    /*
     * Destroy objects
     */
    lcd_cap_delete(b);
    lcd_cap_delete(e);
    /*
     * Exit environment
     */
    lcd_exit(0);
}

Figure 3: Interface example. Return value checking and other details have been omitted.

The module loading code still uses the host page allocator (via vmalloc) to load the kernel module into the host, rather than using the page allocation routines that are part of the microkernel interface. As a result, the pages that contain the kernel module are initially not in the kLCD’s cspace. To handle this, we provide two kLCD-specific routines, klcd_add_page and klcd_rm_page, that add these pages to the kLCD’s cspace and hence bring them into the microkernel’s capability system. Unlike pages allocated via the microkernel interface, these pages are tagged with a special type that prevents them from being freed when the last capability to them is deleted; the kLCD is responsible for freeing them. (Alternatively, the kLCD could allocate pages using the microkernel interface and copy the data over, but this leads to extra overhead. We anticipate that there may be other resources in the future that cannot be duplicated and must be manually added to the capability system.) Note that vmalloc allocates at the granularity of pages, so we don’t need to worry about neighboring host memory being visible inside the LCD.

6. Isolated Threads/LCDs

6.1 Hardware VM Setup

We run isolated kernel modules inside an Intel VT-x hardware virtual machine configured for a 64-bit environment with the address spaces shown in Figure 4. The module is mapped at the same location in the guest virtual address space as it was mapped in the host (in the high address range) so that we don’t need to relocate symbols in the module. The thread that runs inside the LCD uses the 4 KB stack, and we put the message registers at the bottom of the stack. The code creating the LCD can map up to 16 KBs of boot information that the LCD can use to bootstrap its environment. The library kernel uses 20 MBs for managing the guest virtual address space and allocating memory, as we discuss below. We don’t map the first megabyte of guest physical or virtual memory so that we can easily catch null pointer dereferences and because code we isolate in the future may expect BIOS or device memory to be mapped there. Except for the kernel module, all other parts of the guest physical address space are identity mapped in the guest virtual address space.

Like KVM, we use host kernel threads to run the code inside the VM to avoid writing our own scheduling code and so that LCDs have little impact on the performance of non-isolated code. Weibin Sun adapted code from KVM and the Dune project for our needs and completed the first version of the code for setting up a hardware virtual machine. I separated the architecture-independent parts of Weibin’s code into a separate piece of code and simplified some of the virtual machine configuration. The isolated code currently runs without some critical data structures that are needed for more complex code, like the global descriptor table (GDT), interrupt descriptor table (IDT), and task state segment (TSS). All exceptions and interrupts cause a kernel thread running in a virtual machine to exit and return to the non-isolated address space and host kernel environment. Microkernel system calls inside the VM are implemented using the VMCALL instruction.

Proper set up of the code and data structures for Intel VT-x can be difficult and error prone. For example, if
the virtual machine is configured with a non-canonical stack pointer in its VMCS\(^2\), entry into the virtual machine fails, and an uninformative exit code simply indicates ‘bad guest state’. After hours of debugging, I wrote code that implements the checks outlined in the Intel manual that are performed before a virtual machine is entered.

It’s also necessary to keep track of four different address spaces – host physical, host virtual, guest physical, and guest virtual. Without proper care, a programmer can misinterpret an unsigned long host virtual address as a guest virtual address. To prevent these errors, I created four struct data types for each address space type. The address space is now made explicit by the type and checked by the compiler.

6.2 LIBLCD

Since the isolated code runs in a different address space from the host kernel, it has by default no support for common operations like memcpy, memory allocation, console output, and so on. We built a small library kernel, LIBLCD, that implements the isolated version of the microkernel interface and includes a page allocator, slab allocator, cptr cache, console, and library routines like memcpy. LIBLCD is linked with the isolated code and becomes a part of the kernel module that is loaded inside the hardware VM. LIBLCD initializes the cptr cache, page allocator, and slab allocator when the isolated code executes lcd_enter, and it uses information in the boot info pages to bootstrap some of the data structures.

The memory management code was the most challenging part. We built our own simple page allocator that uses bitmaps to track allocation of pages for guest virtual paging structures and pages in the heap. The page allocator uses the microkernel interface to allocate host pages and map them in the LCD’s address space. Since the microkernel only understands cptrs, the page allocator has to track the correspondence between pages and cptrs. We then borrowed the Linux slab allocator and adapted it to use our page allocator and run inside the VM, using some preprocessor tricks and macros.

We used a similar approach to borrow common library routines from the Linux kernel, like memcpy and sprintf, that are needed by the memory management and console code. We developed some techniques for

---

\(^2\)A logical address is canonical when it is sign extended beyond the maximum allowed address width. VMCS stands for virtual machine control structure and is used to configure and track the state of a VM. It contains many fields and represents many aspects of the state of an Intel chip.
eliding unnecessary variables and functions and ensured we had resolved all dependencies by checking that the final kernel module had no unresolved symbols. We built the code so that it would run single threaded with no per-cpu variables or NUMA.

7. Microkernel Internals
The microkernel consists of the following components:

- architecture-dependent code for setting up the hardware VM, which was described previously in Section 6.1;
- a run-loop for entering and exiting LCDs and handling microkernel system calls;
- a capability subsystem;
- code for IPC

The microkernel is roughly 7,000 lines of code, about half of which is for setting up the hardware VM. This count does not include the interface implementations, like kLIBLCD. We briefly describe some components of the microkernel here.

The microkernel uses the host page allocator when kLCDs and LCDs allocate host pages. For example, when an LCD invokes the microkernel system call to allocate a page, the microkernel allocates a zeroed-out host page and inserts it into the LCD’s cspace; the LCD can then map the page in its guest physical address space using a separate system call.

The microkernel also relies on the host scheduler to manage threads that run in LCDs. When an LCD is created, the microkernel creates a host kernel thread that will run inside the LCD. When the LCD is booted, the host kernel thread is started; the thread runs through some initialization steps and then enters the hardware VM.

As explained in the overview, we followed seL4’s design for our capability subsystem, and we implemented seL4-style synchronous endpoints. Muktesh Khole developed the first version of the capability subsystem, and I integrated it into the microkernel by adding code for updating the state of LCDs and the microkernel when a capability to an object is deleted and code for tearing down a microkernel object when the last capability to it is deleted. For example, when an LCD’s capability to a page is revoked, the microkernel should ensure the page is no longer mapped in the LCD’s guest physical address space.

Unlike seL4, we decided not to allow LCDs to have fine-grained control over some parts of their environment, like their guest physical address space. The guest physical address space in Intel VT-x uses a four-level hierarchy of Extended Page Tables (EPTs). To be correct, the microkernel would need to track how these page tables are linked together and properly update them when capabilities to some page tables are deleted. The seL4 team came up with solutions for the 32-bit, single-threaded, two-level paging architectures that seL4 runs on; but we didn’t think these solutions would generalize to a multithreaded, 64-bit architecture.

For simplicity, we also did not implement seL4’s fine-grained allocation of microkernel objects (memory untype-retype) and we do not allow LCDs to have fine grained control over their csaces. There are currently no limits on the amount of memory an LCD can allocate, including the memory allocated on its behalf for microkernel objects. This is obviously a security vulnerability and could be improved in future work, but there could be some big challenges adapting seL4’s design for a multithreaded environment. An LCD’s cspace is limited in size by some microkernel configuration parameters, and csaces are not shared as they are in seL4.

8. Example LCDs
While we have not isolated unmodified kernel code, we have developed some test LCD modules that exercise all parts of the interface, microkernel, and execution environments.

9. Related Work
Researchers have decomposed Linux before, but they put a bare metal microkernel below Linux, and some only coarsely decomposed Linux. Gefflaut, et al. [8] decomposed Linux into user-level servers, but the servers are managed by the L4 microkernel rather than a simple embedded microkernel. Nikolaev and Back [11] ‘vertically’ decomposed Linux into service VMs managed by the Xen hypervisor; each service VM contains a driver, or code that handles system calls that target a specific large subsystem like the TCP/IP stack. But they did not allow for service VMs to communicate with each other, so some system calls are not supported; and they didn’t isolate smaller modules like a specific filesystem. Colp, et al. [4] coarsely decomposed the hypervisor-specific parts of the control VM in the Xen hypervisor into service VMs, but they didn’t decompose traditional subsystems like the network or storage stacks.

---

3What we call a ‘capability derivation tree’ (CDT) in our code is actually different from the capability derivation tree in seL4. Ours would more accurately be called ‘rights derivation tree’.
Others have isolated Linux drivers, rather than large subsystems, by placing them in a separate address space and having them communicate with the core kernel using message passing. Swift, et al. [13] did this, but the driver code still executes in supervisor mode, so it has to be trusted. Boyd-Wickizer and Zeldovich [3] and Ganapathy, et al. [7] moved device drivers into user-level processes that communicate with a stub kernel driver using message passing.

Finally, some researchers have decomposed kernels into smaller, independent modules, but did not isolate them. Kantee [10] decomposed NetBSD into modules that can be linked with other kernel and application code and executed in a user-level process. Ford, et al. [6] developed a framework for re-using arbitrary code from multiple kernels, rather than decomposing only one kernel; but they faced similar challenges like handling module dependencies and setting up interfaces for using the modules.

10. Conclusion

We have constructed a proof-of-concept system that demonstrates that Linux kernel modules can be installed inside minimal hardware VMs and communicate using synchronous IPC. We found along the way that a non-trivial execution environment is needed inside the VMs to support the isolated code so that it can do things like memory allocation and console printing. We also found that fine-grained access control is hard on a modern, multithreaded 64-bit architecture, because the microkernel has to do a lot of bookkeeping in a multithreaded environment, and this leads to memory and synchronization overhead. In the future, we plan to find a more practical level of access control. For example, LCDs may be allowed to allocate a large chunk of memory that is associated with a single capability.

References


