Lightweight Capability Domains: Towards Decomposing the Linux Kernel

Charles Jacobsen   Muktesh Khole [*]   Sarah Spall   Scotty Bauer   Anton Burtsev

University of Utah
Salt Lake City, UT, USA

[*] Microsoft Corporation
Redmond, WA, USA

charlesj@cs.utah.edu   muktesh.khole@utah.edu   spall@cs.utah.edu   sbauer@eng.utah.edu   aburtsev@cs.utah.edu

Abstract

Despite a number of radical changes in how computer systems are used, the design principles behind the very core of the systems stack—an operating system kernel—has remained unchanged for decades. We run monolithic kernels developed with a combination of an unsafe programming language, global sharing of data structures, opaque interfaces, and no explicit knowledge of kernel protocols. Today, the monolithic architecture of a kernel is the main factor undermining its security, and even worse, limiting its evolution towards a safer, more secure environment. Lack of isolation across kernel subsystems allows attackers to take control over the entire machine with a single kernel vulnerability. Furthermore, complex, semantically rich monolithic code with globally shared data structures and no explicit interfaces is not amenable to formal analysis and verification tools. Even after decades of work to make monolithic kernels more secure, over a hundred serious kernel vulnerabilities are still reported every year.

Modern kernels need decomposition as a practical means of confining the effects of individual attacks. Historically, decomposed kernels were prohibitively slow. Today, the complexity of a modern kernel prevents a trivial decomposition effort. We argue, however, that despite all odds modern kernels can be decomposed. Careful choice of communication abstractions and execution model, a general approach to decomposition, a path for incremental adoption, and automation through proper language tools can address complexity of decomposition and performance overheads of decomposed kernels. Our work on lightweight capability domains (LCDs) develops principles, mechanisms, and tools that enable incremental, practical decomposition of a modern operating system kernel.

Categories and Subject Descriptors D.2.12 [*Software Engineering*]: Interoperability—interface definition languages; D.4.6 [*Operating Systems*]: Security and Protection; D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Security

Keywords decomposition, Linux, microkernels

[*] Muktesh Khole contributed to the research described herein while he was a graduate student at the University of Utah.

The definitive version was published in *Proceedings of the 7th Workshop on Programming Languages and Operating Systems (PLOS)*, Farmington, PA, Nov. 2013, `http://dx.doi.org/10.1145/2818302.2818307`

1. Introduction

Over the last decade, attacks on computer systems have undergone major changes in terms of exploit discovery tools, attack complexity, and targeted parts of the system. Attackers routinely use fuzzing tools [42, 43], and attack every possible layer of the kernel: device drivers, the network stack, the stack of USB protocols, file systems, and virtual machine monitors. Even though a number of static [2, 10, 18] and dynamic (stack guards [11], address space randomization [32], executable space protection, control flow integrity [1, 19], code integrity checks [48]) mechanisms have been invented to protect execution of the operating system kernel, attackers come up with new ways to bypass these protection techniques [32, 45, 46, 49]. Modern kernels remain vulnerable [8].

Lack of isolation implies that in a modern system, an attacker is one kernel vulnerability away from taking control over the entire machine. A successful attack on any of the kernel subsystems provides the ability to make the threat persistent in the face of reboots, conceal it from the user and anti-virus security tools and establish a platform for compromising local applications, collecting sensitive financial information and user credentials, mounting attacks on the network hosts, and establishing a distributed, peer-to-peer command and control infrastructure. In 2014, the Common Vulnerabilities and Exposures database lists 133 Linux kernel vulnerabilities that allow for privilege escalation, denial-of-service, and other exploits [13]. This number is consistent across several years [8, 12].

Why do we run monolithic kernels? The reason is twofold. First, for many years, isolation was prohibitively slow due to architectural limits of uniprocessor machines. While significant progress in understanding the costs of inter-process communication mechanisms has been made (see [4] for a historical overview), context switching on commodity hardware remains prohibitively expensive [36, 50] (some embedded CPU architectures are exceptions [4]). For many years, monolithic kernels remained the only practical design choice for performance. Second, the complexity of a monolithic, shared-memory kernel 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. Several attempts to decompose the kernel code failed due to a lack of proper abstractions and automation tools [23, 52].

We argue that, despite all odds, modern kernels can be decomposed. First, decomposition is motivated by recent hardware progress. Today, we run multi-core CPUs with integrated memory and PCIe bus controllers, large last level caches, and low-latency network and storage interfaces. Overheads of traditional kernels already dominate performance of I/O and compute intensive applications [3, 37, 38, 44]. The need for unmediated, low-latency access to both network and PCIe-based flash storage devices, exclusive, untinerrupted access to CPU cores, the ability to minimize the number of cache misses and memory accesses, access to receive packet and flow steering mechanisms, and the ability to avoid overheads of context switches and scheduling motivate some form of decomposition, or at least separation between control and data planes [3, 44] in a commodity kernel. Second, after several decades of engineering effort aimed at modularization of kernel components, kernel subsystems are less tightly coupled, with relatively clean interfaces, and most complexity encapsulated inside individual subsystems. With proper language tools, and a general approach to breaking the code apart, decomposition into isolated subsystems is feasible.

We develop Lightweight Capability Domains (LCDs), a general framework for decomposing the Linux kernel. We make the following contributions. First, to enable practical incremental decomposition, we extend a commodity kernel with a general embedded microkernel interface. This interface enables an execution of decomposed subsystems side by side with the rest of the non-decomposed kernel, and provides a convenient execution environment for developing decomposed subsystems. Second, we develop *decomposition patterns*—a collection of general principles and abstractions for decomposing common patterns of code in the kernel. Third, we design an interface definition language (IDL) aimed at an automatic decomposition effort with minimal changes to the existing kernel code. The IDL is designed for emulating a shared-memory, monolithic environment on top of share-nothing isolated susbsystems. Finally, we make decomposed environments efficient on modern hardware: We combine fast cross-core aysnchronous communication with a lightweight execution model that supports composable asynchronous programming for unmodified kernel code.

2. Decomposition Strategy

An attempt to decompose an entire system at once is an effort that will result in rapidly aging, obsolete code. Instead, we focus our work on developing a platform that 1) enables incremental decomposition isolating one kernel subsystem at a time, 2) can be applied to a rapidly evolving kernel code base, and 3) provides a foundation for developing practical, efficient systems.

Incremental decomposition To provide a path for incremental decomposition and enable execution of legacy monolithic code and isolated subsystems side by side, we embed a small microkernel inside the commodity Linux kernel (Figure 1). The LCD microkernel is similar to the KVM virtual machine monitor embedded into the Linux kernel. Our goal however, is to provide a more general interface suitable for development of semantically rich decomposed subsystems instead of virtualizing a low-level interface of the CPU and I/O devices. The microkernel implements a minimal interface: threads, synchronous communication, capability access control, and memory management.

Breaking the code apart We develop a set of *decomposition patterns*—design and development principles aimed at breaking typical patterns of existing monolithic code into isolated subsystems. Data structures and the code of a commodity monolithic kernel are designed to run in a shared memory environment. The kernel shares control information and state of its components by passing references to objects across subsystems. In a decomposed environment, each kernel subsystem operates on its own version of the system state. This state is synchronized upon cross-subsystem invocations. We develop techniques for transparently synchronizing objects and maintaining a view of a global state on top of isolated, share-nothing domains. Our decomposition patterns cover the commonly used patterns of kernel code: global variables, exported and imported functions, function pointers, data structures, and hierarchies of data structures.

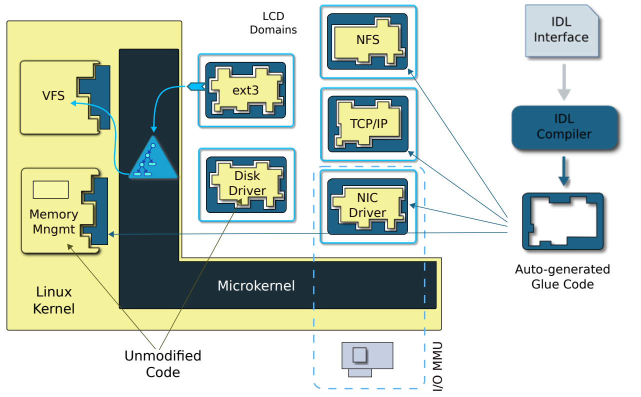

Figure 1. LCDs architecture. LCDs run as a microkernel embedded into the Linux kernel. Isolated subsystems communicate through a capability controlled IPC mechanism.

Interface definition language Decomposition patterns provide a general way to break monolithic code apart. To automate this task, we rely on an interface definition language (IDL) designed to generate cross-domain function invocation and object synchronization code. Our goal for the IDL compiler is to make sure that the majority of code related to crossing the boundaries of isolated domains is automatically generated, and is backward compatible with unmodified legacy code, i.e., isolated subsystems do not require major modifications to their code, and the same code can run in both monolithic and decomposed configurations. The IDL describes interfaces across LCD domains and non-isolated parts of the kernel. Two goals drive design of LCDs's IDL compiler. First, the code generated from the IDL is designed to be compatible with the kernel source: we generate proxy and stub code for kernel functions while preserving function signatures so that it can be linked with the code of unmodified kernel modules and later loaded inside isolated domains. Second, LCDs's IDL provides explicit support for specifying how kernel data structures are synchronized across isolated subsystems.

Capability access control Our motivation for capabilities is twofold. First, we use capabilities to selectively limit authority of isolated subsystems. Each isolated subsystem possesses the smallest subset of rights required to accomplish its task. Thus, the effect of the compromise of an individual kernel subsystem is restricted to the set of resources that the subsystem can access. LCDs borrow ideas from object capability languages [51] and capability microkernels [16]. In LCDs, a capability is an entry in a microkernel-protected data structure, which can be referenced from isolated code via a local name. Inside the microkernel, each capability describes one of the objects implemented by the microkernel or the Linux kernel. Second, in LCDs capabilities implement a notion of cross-domain pointers allowing us to securely reference objects across isolated domains. Similar to the LCD microkernel, each isolated domain implements capability address spaces for the objects it manages, e.g., a file system domain resolves file objects through capabilities. At this level, capabilities allow isolated domains to reference objects inside other domains for which they have authority.

Composable asynchronous I/O Traditionally, kernels use threads as the main execution model and synchronous procedure calls as the main communication mechanism. A single thread of execution moves across kernel subsystems through a series of synchronous function invocations. Control information and data are passed by reference across subsystems as a hierarchy of globally shared data structures. Unfortunately, synchronous function calls do not work well in a decomposed environment. Despite many improvements in understanding synchronous IPC design [4], the hardware context-switching mechanisms did not get faster. Synchronous function invocations are still prohibitively slow. We design LCDs around the idea of CPU cross-core LCD invocations and an asynchronous execution model. Each LCD runs on a dedicated set of CPU cores. We design and build an efficient asynchronous cross-core communication mechanism, and emulate traditional synchronous function invocations on top of asynchronous message passing. To integrate blocking asynchronous messages with existing synchronous code, LCDs implement a cooperative execution environment capable of context switching asynchronous threads implemented as native code.

3. LCDs Architecture

3.1 LCDs microkernel

The LCD microkernel follows the design of the L4 microkernel family [17]. Two parts of the microkernel interface are critical for the LCDs architecture. Similar to seL4 [15], the LCD microkernel implements a pure capability-based IPC that explicitly controls all communication across isolated subsystems. This synchronous IPC is used for requesting microkernel resources, and specifically establishing regions of shared memory for future asynchronous communication. Similar to [34], we implement capability address spaces (`CSpaces`) and capability derivation trees.

We make several pragmatic design choices to simplify development. While demonstrating extremely slow context switching performance [36], we still choose hardware-assisted virtualization for isolation as it is easier to program, e.g., handle low-level hardware conditions inside and outside isolated domains. By developing LCDs as kernel modules, we are able to reuse linking and loading functionality provided by the Linux kernel. 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. We further rely on Linux kernel threads and the Linux scheduler for scheduling of microkernel threads.

3.2 Interface Definition Language

The LCDs IDL compiler is responsible for generating inter-domain communication and synchronization code that ensures transparency of decomposition for the original monolithic code. The core of the IDL are definitions for invoking isolated subsystems (remote function invocations), and disciplines for synchronizing kernel data structures (Section 3.3).

Functions Our main goal is that the IDL compiler generates proxy/stub functions with the same signatures as the functions that are used in a monolithic kernel. For example, to export a function with the following signature:

```
int register_filesystem(...);
```

LCDs require the following IDL definition of an interface:

```
interface filesystem (capability cap = null) {
    rpc int register_filesystem(capability cap, ...);
}
```

The function definition is extended with a capability that explicitly names the rendezvous point or asynchronous channel implementing the interface. To provide compatibility with unmodified code, the interface is declared to have an optional parameter: a capability to the rendezvous point that implements an interface of a virtual file system. The instance of the interface can be declared to take the capability to the interface:

```
interface filesystem (vfs);
```

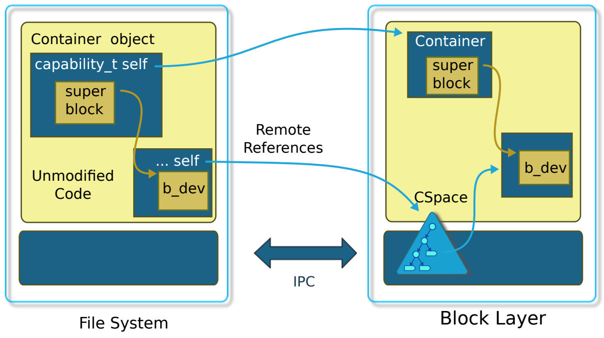

Figure 2. Isolated object hierarchies in LCDs.

From the interface definition and the above declaration, we generate the following backward compatible stub:

```
int register_filesystem(struct file_system_type *fs) {
    return register_filesystem_callee_stub(vfs, fs);
}
```

where the vfs is specified during LCD intialization, which is described below.

Initializing LCDs The IDL provides support for declaring interfaces imported and exported by an LCD. For example, an LCD that is running a file system and requires access to the virtual file system and block device interfaces might use the following configuration file that is supported by the IDL:

```
module fs {
    require capability vfs;
    require capability bdev;
}
```

The IDL generates two functions: one is used by the main kernel to create a new LCD, and the other function initializes the LCD internally during boot.

3.3 Decomposition Patterns

LCDs aim to implement an environment in which isolated subsystems *do not share state* (Figure 2). Instead, multiple isolated subsystems maintain their own private hierarchies of objects and synchronize them explicitly upon function invocations. This assumption is key for security—one compromised subsystem cannot affect execution of others through modifications of shared objects.

Object synchronization A typical pattern in the Linux kernel is to pass objects by reference across subsystems. In case of LCDs, however, decomposed subsystems do not trust each other. To ensure isolation, each LCD maintains a private shadow copy of each object. LCDs allow explicit control over a set of object fields that will be passed and returned upon function invocation with the mechanism of *projections*.

```
// Projecting two fields of the super_block data structure
projection super_block <struct super_block> {
    [out,in] type1 field1;
    [out] type2 field2;
}
rpc type0 foo(projection super_block *sb);
```

A projection explicitly defines a subset of fields of the projected object that will be passed to the callee and returned from it during the domain invocation. The IDL supports scopes so the same object type can be projected differently depending on the function.

Stateful objects and remote object references In the kernel code it is common that two or more subsystems go through multiple steps of a communication protocol that requires that objects are preserved across a series of cross-domain invocations. Both caller and callee domains require access to instances of the same private objects multiple times. We use a mechanism of *remote references* to lookup a specific private object across domain boundaries (Figure 2). In LCDs, remote references are capabilities that point to objects inside other subsystems. Similar to the LCD microkernel, each subsystem uses the mechanism of capability identifiers to protects all objects it exports to its clients. When an object is referred across boundaries of isolated subsystems, a remote reference (or a capability) is resolved by the callee subsystem into a private instance of an object through the mechanism of capability address spaces. To support transparent referencing of objects across domains, every object that exists in multiple subsystems is paired with a capability reference that is used to lookup a corresponding object copy in that subsystem. We rely on the mechanism of *container objects* that encapsulate original kernel data structures to avoid code modifications and leave original kernel data structures unchanged. (Figure 2). A projection definition below uses a bind keyword to specify that the projected super_block object must be resolved using the self capability stored in the container object (referred as parent) in the callee's domain.

```
// Using remote reference to update remote object
projection super_block <struct super_block> {
    [bind, in, alloc] capability container->self;
    [in] type1 field1;
}
rpc int fill_super(projection super_block *sb, ...);
```

The IDL provides alloc, bind, and free keywords to control when remote objects are allocated, looked up, and freed. Similar mechanisms are used for returning objects, i.e., allocating a private copy in the caller domain, and initializing it with initial values.

Object hierarchies It is common that objects contain pointers to other objects. When a root of the object hierarchy is passed as an argument to a cross-domain invocation, the entire hierarchy must be marshaled and reconstructed on the callee's side. LCDs rely on mechanisms of projections that are used to describe marshaling of object hierarchies:

```
// Marshaling a hierarchy of objects with projections
projection block_device<struct block_device> {
    [in] type1 field1;
    [bind] capability container->self;
};
projection super_block <struct super_block> {
    [in] projection block_device *s_bdev;
    [bind] capability container->self;
}
rpc type0 foo(capabilty fs, projection super_block *sb);
```

We support synchronization of simple object hierarchies. In practice, requirements for kernel modularity have already eliminated most of the cases when a more complicated object hierarchy crosses the boundaries of kernel subsystems.

3.4 Fast Communication

Asynchronous cross-core communication LCDs' fast IPC mechanism is designed for efficient notification, and zero-copy transfer of data across isolated subsystems in a multicore environment (Figure 3). In LCDs a caller and callee establish a region of shared memory through slow synchronous invocations. After that, communication is exit-less and does not involve the microkernel. Similar to Beltway Buffers [14], the shared region of memory holds multiple kinds of buffers and communication rings. Data buffers are used to hold bulk data, e.g., payload of network packets. Communication

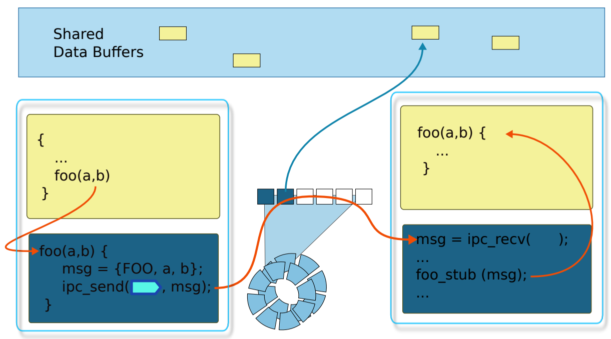

Figure 3. Fast asynchronous IPC in LCDs.

rings serve as lock-free message queues for sending and receiving messages. To achieve zero-copy transfer of bulk data, messages contain pointers into shared data buffers.

We optimize a cross-core messaging protocol for optimal utilization of the hardware cache coherency protocol. Each message channel consists of two rings (outgoing and incoming messages). Similar to FastForward [24], we avoid shared producer and consumer pointers, as they cause expensive cache contention on every update. Instead, we utilize an explicit `state` flag that signals the end of available messages, and free slots. As we aggressively optimize the number of cache transactions involved in each message transfer, we configure each message to be a size of a single cache line (64 bytes on our hardware). We use modulo two arithmetic to avoid expensive division operations on the IPC path. Finally, we rely on ring polling on both sender and receiver, however the polling is integrated into our cooperative asynchronous execution environment (Section 3.5). We use `monitor/mwait` instructions instead of polling when sender and receiver become idle.

Inside a single CPU core, modern CPUs implement a version of a directory based protocol in which the directory is maintained by the shared last level cache. The directory maintains information about the state of every cache line in the cache hierarchy of every core of a single CPU. A snooping cache coherence protocol is used to synchronize cache lines across CPUs. Two cache transactions are involved in each part (send and receive) of the IPC. First, the caller tries to update the cache line sending a message. Its L1 cache issues a request for ownership to the cache directory. To serve this request, the directory needs to contact the L1 cache of the callee core and invalidate the cache line. The directory then replies to the caller's L1 cache changing the state of the line from "invalid" to "modified", and allowing update of the line with the new message. Second, immediately, after the cache line is updated by the caller, it is re-fetched from the callee's busy-wait loop. To read the cache line, the callee issues a similar transaction to the cache directory that in turn reaches the L1 cache of the caller. Based on the BenchIT CPU benchmark analysis [40, 41]) each of the above transactions that involves two L1 caches on the same core requires around 80 cycles. This theoretical analysis matches performance of our IPC implementation (Table 1). We perform a send/reply message sequence that requires four cache transactions in 384 cycles on our Nahelem Intel Xeon CPU E5530 clocked at 2.40GHz. In addition to the cache coherency overheads, we lose around 12-16 cycles in send and receive code. Our IPC code is written in C with minor addition of manual assembly, e.g., a `pause` instruction that saves around 88 cycles in a busy-wait loop. A full call/reply invocation of a void function takes 432 cycles (marshaling of six unsigned long parameters adds 4 cycles). The overhead of using monitor/mwait on one of the ends of the channel is 342 cycles. When multiple outstanding messages are queued in the ring buffer, the costs of the

Test	Cycles (ns)
Send/receive message	384 (160)
Send/receive message (queue of 4)	159 (67)
Call/reply invocation (void function)	432 (180)
Call/reply invocation (6 arguments)	436 (182)
Overhead of mwait (receiver enters halt state)	726 (303)

Table 1. Achieved cross-core IPC performance.

cache coherence protocol goes down. On a queue length of four, a send/receive sequence takes only 159 cycles.

3.5 Composable Asynchronous Communication

Synchronous procedure calls do not compose well with blocking, asynchronous communication mechanisms. If a thread sends an IPC message (e.g., implementing a remote procedure call into another isolated subsystem), it needs to wait for a reply from the callee. Several strategies are possible. First, the caller can remain waiting in a busy wait loop. If the invocation is fast the caller can reply nearly instantly. But in a typical scenario remote invocations take thousands of cycles. Second, the system can rely on traditional thread context switching to utilize the CPU. Unfortunately, traditional thread switches are expensive due to an expensive exit into the microkernel. Alternatively, the code of the system can be designed to implement an asynchronous programming model [26, 35].

With LCDs we aim to 1) avoid expensive context switches, and 2) provide backward compatibility with existing kernel code, i.e., avoid reimplementing kernel code in a message friendly manner. We implement a composable asynchronous I/O without stack ripping that relies on a lightweight context switch implemented as a form of cooperative thread scheduling. We base our implementation on AC [26], but change it to work inside the Linux kernel. Similar to AC, we leverage functionality of GCC macros and nested functions to avoid compiler modifications.

3.6 `libKernel` Environment

`libKernel` Decomposed subsystems run in isolation from the rest of the kernel. However, the logic of any kernel subsystem depends on a set of common primitives provided by the kernel environment: memory allocation, synchronization, string and memory copy, console output, etc. We built a small library kernel, `libKernel`, that implements a minimal instance of the kernel, and includes page and slab allocators, capability management code, console, and common kernel utilities like memcpy. `libKernel` is linked with the isolated code and becomes a part of the kernel module that is loaded inside the isolated domain. We implement a simple page allocator that uses the microkernel interface to allocate host pages and map them in the LCD's address space. Since the microkernel only understands capability identifiers, the page allocator has to track the correspondence between physical pages and capability identifiers. Using preprocessor macros, we adapted the Linux slab allocator to run inside LCDs. We use a similar approach to borrow common library routines from the Linux kernel, like memcpy and sprintf. We developed techniques for eliding unnecessary variables and functions and ensured we had resolved all dependencies by checking that the final kernel module had no unresolved symbols.

4. Related Work

The concept of decomposing operating system services for isolation and security is not new [23, 27]. Multiple projects try to apply this principle in practice in both microkernel [5, 20, 29–31, 33] and virtual machine [7, 22, 39, 47, 53] based systems. Most notable, SawMill was a research effort performed by IBM aimed at building a decomposed Linux environment on top of the L4 microkernel [23]. SawMill was an ambitious effort to decompose the entire kernel at once. Unfortunately, the project became suspended due to organizational politics and untimely decease of one of its leaders, Jochen Liedtke. Nooks was primarily aimed at isolation of device drivers inside the Linux kernel [52]. While lacking an explicit interface definition language, Nooks developed mechanisms for generating entry points and skeletons for the glue code from the kernel header files. Similar to LCDs, Nooks maintained and synchronized private copies of kernel objects, however, this synchronization code had to be developed manually. OSKit developed a set of decomposed kernel subsystems out of which a full-featured OS kernel could be constructed [21]. 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 COM component object model. OSKit quickly became outdated and unsupported.

Existing hypervisors succeed in providing complete isolation of applications at the level of hardware virtualization, but they provide no mechanisms to support decomposition of kernel subsystems. Multiple commercial projects attempt to secure commodity applications using full-system virtualization (Qubes OS [47], Bromium [7], XenClient [53]). Complete isolation works well when no sharing is required, e.g., in case of individual applications or device drivers [6]. An untrusted desktop application is sealed in its virtual container, and runs until it exits [7, 47, 53]. However, core operating system services which are inherently designed to provide sharing of resources (e.g. file systems, block storage, network stack), require mechanisms for sharing and collaboration to avoid operating with reduced functionality or with excessive privilege. Several military-grade certified secure virtualization projects apply the principle of secure isolation to traditional systems [33, 39]. The proprietary nature of these systems limits their impact in the broader community.

Do we have to decompose existing kernels or is it better to re-implement decomposed environments from scratch? Multiple projects attempt to re-implement kernel functionality from scratch in a safer, verification friendly language [9, 25, 28, 34, 54]. Although promising, these approaches are still far from being applicable in a realistic deployment. Modern kernels accumulate several decades of development effort that result in irreplaceable functionality: hundreds of device drivers, dozens of network protocols, block storage stacks, file systems, and CPU and I/O schedulers. To be practical, decomposition must aid security and reliability of existing OS kernels and become an integral part of the kernel development process with low overhead for developers.

5. Conclusion

Decomposition of commodity kernels has remained an unsolved problem for decades. We believe LCDs provides a practical step towards solving this problem. A combination of fast cross-core communication, asynchronous execution model, general decomposition patterns, and language support result in a decomposition platform that enables practical, efficient, and secure systems. LCDs is an evolving platform. In the future, we intend to extend it with a static analysis aimed at automatic generation of interface and projection definitions. We further plan to use LCDs as a practical platform for enabling verification of commodity operating systems.

Acknowledgments

We thank Weibin Sun, Jon Rafkind, and the anonymous PLOS reviewers. This material is based upon work supported by the National Science Foundation under Grant No. 1319076, Google, and NetApp.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity - principles, implementations, and applications. In *CCS*, 2005.

[2] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, 2014.

[4] B. Blackham and G. Heiser. Correct, fast, maintainable: choose any three! In *APSys*, page 13, 2012.

[5] 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.

[6] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX ATC*, pages 9–9, 2010.

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

[8] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys*, pages 5:1–5:5, 2011.

[9] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, et al. SAFE: A clean-slate architecture for secure systems. In *Technologies for Homeland Security (HST)*, pages 570–576, 2013.

[10] Coverity, Inc. Coverity SAVE, 2012. `http://www.coverity.com/products/coverity-save.html`.

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, and J. Walpole. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

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

[13] CVE Details. Vulnerabilities in the Linux kernel in 2014. `http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/Linux-Linux-Kernel.html`.

[14] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the OS traffic jam. In *INFOCOM*, 2008.

[15] P. Derrin, D. Elkaduwe, and K. Elphinstone. seL4 reference manual. Technical report, ERTOS NICTA. `http://www.ertos.nicta.com/research/sel4/sel4-refman.pdf`.

[16] D. Elkaduwe. *A principled approach to kernel memory management*. PhD thesis, University of New South Wales, 2010.

[17] K. Elphinstone and G. Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *SOSP*, pages 133–150, 2013.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–1, 2000.

[19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, pages 75–88, 2006.

[20] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture*. Techn. Univ., Fakultät Informatik, 2007.

[21] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *SOSP*, pages 38–51, 1997.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP*, pages 193–206, 2003.

[23] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. 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.

[24] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP*, pages 43–52, 2008.

[25] Gu, L., Vaynberg, A., Ford, B., Shao, Z., and Costanzo, D. CertiKOS: a certified kernel for secure cloud computing. In *APSys*, page 3, 2011.

[26] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: composable asynchronous IO for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

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

[28] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, 2014.

[29] 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.

[30] 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.

[31] 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.

[32] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS*, pages 298–307, 2004.

[33] INTEGRITY Real-Time Operating System. `http://www.ghs.com/products/rtos/integrity.html`.

[34] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., and others. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[35] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX ATC*, pages 7:1–7:14, 2007.

[36] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

[37] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni. Architectural breakdown of end-to-end latency in a TCP/IP network. *Int. J. Parallel Program.*, 37(6):556–571, Dec. 2009.

[38] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[39] LynuxWorks. Desktop virtualization and secure client virtualization based on military-grade technology.

[40] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Workshop on Memory Systems Performance and Correctness*, pages 4:1–4:10, 2014.

[41] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*, pages 261–270. IEEE, 2009.

[42] Moritz Jodeit and Martin Johns. USB device drivers: A stepping stone into your kernel. In *European Conference on Computer Network Defense*, 2010.

[43] T. Mueller. Virtualised USB fuzzing for vulnerabilities. 2010. `https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf`.

[44] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy,

T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, 2014.

[45] Bypassing StackGuard and StackShield. Phrack Magazine. Volume 0xa. Issue 0x38.

[46] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012. `http://doi.acm.org/10.1145/2133375.2133377`.

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

[48] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, pages 335–350, 2007.

[49] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561, 2007.

[50] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8, 2010.

[51] M. Stiegler. The E language in a walnut, 2000. `http://www.skyhunter.com/marcs/ewalnut.html`.

[52] M. M. Swift, S. Martin, H. M. Levy, and S. 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.

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

[54] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Notices*, volume 45, pages 99–110. ACM, 2010.