Lightweight Capability Domains: Decomposing the Linux Kernel

Charles Jacobsen *  Sarah Spall *  Muktesh Khole *  John Regehr  Anton Burtsev
University of Utah
{charlesj, sspall, mkhole, regehr, aburtsev}@cs.utah.edu

In a modern system, an attacker is one kernel vulnerability away from taking control over the entire machine. Despite a number of advances in program analysis, software testing, and verification, serious bugs still appear in modern kernels. From January to August 2014, the Common Vulnerabilities and Exposures database lists 72 Linux kernel vulnerabilities that allow for privilege escalation, denial-of-service, and other exploits. It is unlikely that the large, semantically rich functionality of traditional operating system services can be effectively secured within a monolithic kernel. Modern kernels need isolation as a practical means of confining the effects of individual attacks.

Why do we run monolithic kernels? The reason is twofold. First, for many years, isolation was prohibitively slow due to context switching on uniprocessor machines. Monolithic kernels were the only practical design choice for performance. Second, complexity of monolithic kernels prevents trivial decomposition efforts. Decomposition of a modern full-featured kernel requires cutting through a number of tightly-connected, well-optimized subsystems that use rich interfaces and complex interaction patterns.

We argue that, despite all odds, modern kernels can be decomposed. Careful choice of communication abstractions, a general approach to decomposition, a path for incremental adoption, and automation through proper language tools can address the complexity of decomposition. Our work on lightweight capability domains (LCDs) develops principles, mechanisms, and tools that enable incremental decomposition of modern operating system kernels.

Incremental decomposition. To be practical, decomposition must be an incremental effort that isolates one subsystem at a time. LCDs enable execution of legacy monolithic code and isolated subsystems side by side by embedding a microkernel interface inside the OS kernel. The microkernel implements a small set of interfaces, communication, and synchronization mechanisms that allows us to place entire subsystems or a single function from the core kernel inside an isolated protection domain, while other parts of the kernel are left untouched. Moreover, by relying on general decomposition patterns and language tools, we make sure that the majority of code related to crossing isolation boundaries is automatically generated, and backward compatible. In other words, isolated service implementations do not require major modifications to their code, and the same code can run in both monolithic and decomposed configurations.

Breaking the code apart. Data structures and the code of a modern kernel are designed to run in a shared memory environment. The kernel heavily shares control information and state of its components by passing references to objects across subsystems. In a decomposed environment, each subsystem operates on its own version of a system state. This state is synchronized upon cross-subsystem invocations. We develop a set of decomposition patterns—design and development principles aimed at breaking typical patterns of existing monolithic code into isolated subsystems. Our work is feasible due to several decades of engineering effort aimed at modularization of kernel components. Kernel subsystems are loosely coupled, with relatively clean interfaces, and most complexity encapsulated inside individual subsystems.

Capability access control. In LCDs, capabilities serve as a general foundation for constructing least privilege services out of existing components of the traditional operating system stack. Each component 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 and capability microkernels. 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. Capability names are the only means to reference and exchange resources across protection domains. Finally, capabilities implement a notion of cross-domain pointers allowing us to securely reference objects across isolated domains, and also virtualize the resource space enabling execution of multiple copies of kernel subsystems.

Rich interface definition language. LCDs rely on a powerful interface definition language aimed at automation of the decomposition effort and reuse of unmodified kernel code. The IDL is used to describe interfaces of domains and non-isolated parts of the kernel that domains can use, and will allow for describing stateful interaction patterns and how objects are synchronized across subsystems. We develop an IDL compiler that generates low-level C code that uses the microkernel primitives. This glue code is built and linked with the domain’s and non-isolated kernel’s code so that interactions between domains and non-isolated parts of the kernel are transparent.

Control and data paths. Despite many improvements in synchronous IPC design, synchronous function invocations are still prohibitively slow across isolated domains. We balance complexity of decomposition and performance by separating control and data paths in the kernel code. Control paths will remain slow, synchronous, but unmodified. Data paths will rely on fast asynchronous communication primitives but will require changes to its code.

* Students. We do not plan a demo.