

Technical Perspective

‘What Is the Ideal Operating System?’

By Daniel Lohmann

MY OPENING QUESTION for oral exams is an icebreaker for nervous students because no answer is wrong. It always depends on the application.

Operating systems (OSs) provide no business value on their own. Their *sole* purpose is to ease the development, integration, and operation of applications—that is, to provide the “right” set of abstractions and policies (and map them efficiently to the underlying hardware) for a particular application use case. The application use case may be your general-purpose desktop computer, an embedded real-time system, or your business service running in the cloud. The ideal OS provides exactly what is needed for *your* application—but nothing more.

Fulfilling the what-is-needed part, that is, the functional requirements, has become relatively easy. Linux, for instance, supports about 30 different hardware architectures and application domains from embedded real-time systems up to ultra-scale servers. It is the nothing-more part (a nonfunctional requirement) that is challenging. The enormous versatility of modern OSs comes at the price of a significant code and memory bloat: Approximately 50%–80% of the OS code remains unused. Even though many users tend to not care about a few MiB of RAM and a few GiB of disk space taken by cruft (“RAM is cheap. Disks are even cheaper.”), this nevertheless comes at a price:

- *Bloat scales.* What may appear negligible for a single system leads to significant hardware and energy costs for cloud providers, who host thousands of these systems. Code that is not there does neither prolong boot time nor consume memory or network bandwidth.

- *Increased attack surface.* While you may have no use for feature *X*, an attacker might be more than happy about its presence on your system. Code that isn’t there cannot be abused.

- *Higher maintenance efforts.* Patching your systems early and, thus, way

```
inline void spin_irq_lock(raw spinlock_t *lock) {
    irq_disable();
#ifdef CONFIG_SMP
    spin_acquire(&lock)
#endif
}
```

too often? Code that is not there does not need to be patched.

System software developers are aware of these problems but are caught between the conflicting demands of broad versatility and case-specific efficiency. To overcome this dilemma and make everybody happy, most OSs support a broad range of features and hardware platforms but can be tailored at compile-time with respect to a specific use case, often by means of conditional compilation as shown in accompanying listing.


In Linux, support for symmetric multiprocessing (SMP) is an optional feature and the feature flag `CONFIG_SMP` is used throughout the kernel code (it is said to be an “`#ifdef hell`”) to tailor its implementation for single- or multi-core operation. The Kconfig frontend (just enter “`make menuconfig`”) presents all available features and their dependencies for configuration in a tree-like structure. Hence, you can tailor Linux to provide exactly what is needed for your application—the ideal OS is at your fingertips!

The only thing is Linux already provides more than 17,000 such `CONFIG_` flags—and keeps on growing. So which ones do you need? OS tailoring has not only become a more than tedious task, it also still requires profound expert knowledge. It is understandable that people prefer the include-all standard configuration.

This is where approaches for *automatic* kernel tailoring (and, thus, debloating) come into play. In a nutshell, they first “measure” the features needed by your application while executing on an (instrumented) include-all kernel. In the second step, this information is then aggregated to derive a tailored kernel configuration and build a specialized

kernel for your specific use case. The results are compelling: Code size and attack surface are reduced by 50%–80%, known vulnerabilities by 34%–74%. Nevertheless, even 10 years after becoming available¹ and even though trends like function-as-a-service have led to a massive increase of dedicated VMs running in the cloud, automatic kernel tailoring is still not employed in practice. Why is that the case?

In the following paper, the authors put a fresh view on the *practicability* of automatic kernel debloating. They take the stand of a cloud-service integrator to analyze the shortcomings and obstacles of the existing techniques and overcome them in an easy-to-use tool named COZART. Their main technical contribution, besides an improved approach to detect the required kernel features, is the introduction of composability of platform-specific and application-specific kernel feature sets, which significantly reduces effort when preparing a tailored VM for function-as-a-service scenarios.

However, their paper is of much broader interest, as it also shows us that the (felt) abundance of computing resources has led our discipline to become careless and our software systems to include way too much cruft. We all teach our students how to use and design extensible software systems. But the more challenging part really is to design software that is shrinkable. 

Reference

1. Tartler, R. et al. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *Proceedings of the 8th Intern. Workshop on Hot Topics in System Dependability*, 2012, USENIX Assoc.

Daniel Lohmann is a professor at Leibniz Universität Hannover, Germany.

Copyright held by author.