DOI:10.1145/3524301

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3524299

Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating

By Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu

Abstract

This paper presents a study on the practicality of operating system (OS) kernel debloating, that is, reducing kernel code that is not needed by the target applications. Despite their significant benefits regarding security (attack surface reduction) and performance (fast boot time and reduced memory footprints), the state-of-the-art OS kernel debloating techniques are not widely adopted in practice, especially in production environments. We identify the limitations of existing kernel debloating techniques that hinder their practical adoption, such as both accidental and essential ones. To understand these limitations, we build an advanced debloating framework named Cozart that enables us to conduct a number of experiments on different types of OS kernels (such as Linux and the L4 microkernel) with a wide variety of applications (such as HTTPD, Memcached, MySQL, NGINX, PHP, and Redis). Our experimental results reveal the challenges and opportunities in making OS kernel debloating practical. We share these insights and our experience to shed light on addressing the limitations of kernel debloating techniques in future research and development efforts.

1. INTRODUCTION

Commodity operating systems (OSs), such as Linux and FreeBSD, have grown in complexity and size over the years. However, an application usually requires only a small subset of OS features[22] that highlights the bloat that exists in OS kernels. Such bloat in OS kernels leads to increased attack surfaces, prolonged boot time, and increased memory usage. Application-oriented OS kernel debloating—reducing the kernel code that is not needed by target applications—is reported to be effective in mitigating the above issues. For example, prior work reported that 50%–85% of the attack surface can be reduced by debloating the Linux kernel for the server software[16] and that 34%–74% of known security vulnerabilities can be nullified in the Linux kernel by only including kernel modules that are needed by the target applications.[4]

Recent trends in function as a service and microservices, where numerous kernels are often packed and run in virtual machines (VMs), further strengthen the importance of kernel debloating. In these scenarios, VMs run small applications and each application tends to be "micro" with a small kernel footprint.[3, 11] Some recent virtualization technologies (e.g., LightVM[18]) require users to provide *minimalistic* Linux kernels for target applications.

Debloating OS kernels by hand-picking kernel features is impractical due to the complexity of commodity-off-the-shelf (COTS) OSs.[9, 23, 24] Linux, for example, has more than 14,000 configuration options (as of v4.14) with hundreds of new options introduced every year. Kernel configurators (e.g., KConfig) do not help debloat the kernel but only provide a user interface for selecting configuration options. Given the poor usability and incomplete documents,[9] it is difficult for users to select the minimal kernel configuration.

Recently, several *automated* kernel debloating techniques and tools have been proposed and built.[12, 15–18, 21, 25] Despite their diversity, existing kernel debloating techniques share the same working principle, that is, the following three steps: (1) running the target application workloads and tracing the kernel code that was executed during the application runs; (2) analyzing the traces and determining the kernel code that is needed by the target applications, and (3) assembling a debloated kernel that only includes the code required by the applications.

Configuration-driven (also known as feature-driven) debloating is the de-facto method for OS kernel debloating.[6, 13, 16–18, 21, 25] Most existing tools use configuration-driven debloating techniques as they are among the few techniques that can produce stable kernels. Configuration-driven kernel debloating reduces kernel code based on *features*: a kernel configuration option corresponds to a kernel feature. A debloated kernel *only* includes features for supporting the target application workloads.

However, despite their attractive benefits regarding security and performance, automated kernel debloating techniques are not widely adopted in practice, especially in production use cases. This is not due to a lack of demand—we observe that many cloud vendors (e.g., Amazon AWS, Microsoft Azure, and Google Cloud) handcraft the Linux kernel code to reduce code, which is not as effective as automated kernel debloating techniques (Section 2.4). Our

The original version of this paper was published in *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, May 2020.

goal is to understand the limitations of kernel debloating methods that hinder their practical adoption and share our experience to shed light on addressing these limitations.

1.1. Accidental and essential limitations

We identify five major limitations of existing kernel debloating techniques based on our own experiences of using existing tools and our interactions with practitioners.

- *No visibility during the boot phase*. Existing techniques can only be initiated after the kernel boot and hence cannot observe what kernel code is loaded during the boot phase (mainly due to their reliance on `ftrace`). As a result, the kernel debloated by existing techniques oftentimes cannot boot[6] because critical modules could be missing from the debloated kernel. Our experiments show that up to 79% of kernel features can only be captured by observing the boot phase. Furthermore, existing techniques could miss important performance and security features that are, again, only loaded at boot time (e.g., `CONFIG_SCHED_MC` for multicore support and `CONFIG_SECURITY_NETWORK`), leading to reduced performance and security.
- *Lack of support for fast application deployments*. Using existing tools, deploying a new application on a debloated kernel requires going through the entire three steps of tracing, analysis, and assembling. The process is time-consuming (can take hours or even days) and thus prevents agile application deployment because starting up a new application needs to wait until the generation of the debloated kernel is completed.
- *Coarse granularity*. Existing techniques use `ftrace` that can only trace kernel code at the *function* level. Function-level tracing is too coarse-grained to track configuration options that affect intrafunction code.
- *Incomplete coverage of kernel footprints*. Because kernel debloating uses dynamic tracing, it requires application workloads (encoded as benchmarks) to drive kernel code execution to maximize coverage. However, it is challenging for benchmarks to cover the complete kernel footprints of target applications, and the debloated kernel would likely crash at runtime especially if the application reaches kernel code that was not observed during tracing.
- *Not distinguishing what is executed versus what is needed*. Existing techniques include kernel features that are reached during the application workloads, even though the executed code may not be actually needed, for example, a second filesystem may be initialized but never needed.

To analyze the above limitations, we divide them into *essences* (limitations inherent in the nature and assumptions of kernel debloating) and *accidents* (those that exist in current kernel debloating techniques but that are not inherent to the process). We believe that the first three limitations are accidental and can be addressed by improving the design and implementation of the debloating tools, whereas the last two are essential that require efforts beyond the specific debloating techniques themselves.

1.2. Results and findings

We focus on OS kernels deployed in cloud environments that are typically used as guest OSs to run untrusted applications with high-performance requirements.

We developed COZART, a new OS kernel debloating framework on top of QEMU. COZART employs instruction-level tracing from QEMU to achieve visibility into the boot phase. COZART tracks the kernel code and maps it to kernel configuration options. COZART moves kernel tracing off the critical path—with our framework, one can generate CONFIGLETS (a set of configuration options) *offline* specific to an application or a deployment environment. We term the former APPLETS and the latter BASELETS. Given a set of target applications, COZART can generate a debloated kernel by directly *composing* the APPLETS and the BASELET generated offline into a full kernel configuration. Such composability enables COZART to *incrementally* build new kernels by reusing CONFIGLETS (saving repetitive tracing efforts) and previously built files (e.g., the object files and LKMs).

We use COZART as a vehicle to conduct a number of experimental studies on different types of OS kernels (such as Linux, the L4 microkernel[1], and Amazon Firecracker kernel[2]) with a wide variety of applications (such as HTTPD, Memcached, MySQL, NGINX, PHP, and Redis). Our studies lead to the following results and findings:

- Existing techniques initialize kernel tracing too late and cannot observe the boot phase, which is critical to producing a bootable kernel. COZART uses the tracing feature provided by the hypervisor to obtain end-to-end observability and *produce stable kernels*. Kernels debloated by COZART do not have performance regressions, meanwhile, it achieves a 13+% reduction of boot time and over 4MB memory savings of the systems.
- Kernel debloating can be done within tens of seconds if the configurations of the target applications are known. The composability of COZART allows APPLETS to be prepared offline. To deploy a new application on a debloated kernel, COZART can compose the APPLETS of the target application with the CONFIGLETS of the current kernel to generate a new debloated kernel.
- Using instruction-level tracing (as opposed to `ftrace`) can address kernel configuration options that control intra-function features. The overhead of instruction-level tracing is acceptable for running test suites and performance benchmarks.
- An essential limitation of using dynamic-tracing based techniques is the imperfect existence of test suites and benchmarks. Specifically, the official test suites of many open source applications have *low code coverage*. Combining different workloads (e.g., using both test suites and performance benchmarks) to drive the application could alleviate this limitation to a certain extent.

- Domain-specific information can be used to further debloat the kernel by removing the kernel modules that were executed in the baseline kernel but are not needed during actual deployment. Take Xen and KVM as examples. We can use Cozart to further reduce the kernel size (by 40+% and 39+%) based on the `xenconfig` and `kvmconfig` configuration templates respectively.
- Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and even extensively customized kernels (e.g., the Firecracker kernel). The reduction is significant: 47.0% for L4 and 19.76% for Firecracker.

2. KERNEL CONFIGURATION

We first provide an overview of the kernel configuration ecosystem using Linux as an example.

2.1. Configuration options

A kernel configuration is composed of a set of configuration *options*. A kernel module could have multiple options, each of which controls which code will be included in the final kernel binary. Configuration options control different granularities of kernel code such as statements and functions which are implemented by C preprocessors, as well as object files which are implemented based on Makefiles entries.

C preprocessors select code blocks based on `#ifdef/#ifndef` directives—configuration options are used as macros to determine whether to include such conditional groups in the compiled kernel. Figure 1a and b shows examples of C preprocessors for two configuration options, with statement and function granularity, respectively.

Makefiles are used to determine whether or not to include certain object files in the compiled kernel. For instance, in Figure 1c, `CONFIG_CACHEFILES_HISTOGRAM` and `CONFIG_CACHEFILES` are both configuration options in a Makefile. The detailed characteristics of kernel configuration patterns have been studied before.[7,19]

Statement-level configuration options (such as those in Figure 1a) cannot be identified by function-level tracing used by existing kernel debloating tools.[12,13,16] In fact, we find that 31% of C preprocessors in Linux 4.18 are statement-level options.

The number of configuration options in modern monolithic OS kernels is increasing rapidly as a result of the rapid growth of kernel code and features. Taking Linux as an example, the kernel versions 5.0, 4.0, and 3.0 have 16,527, 14,406, and 11,328 configuration options, respectively.

2.2. Configuration languages

Various OS kernels employ different configuration languages to instruct the compiler on which code to include in the compiled kernel. For example, Linux and L4/Fiasco use KConfig, eCos uses CDL, and FreeBSD uses a key-value format. The languages allow for the definition of configuration options and dependencies between them. Without loss of generality, we use KConfig as an example of kernel configuration semantics throughout the rest of this paper.

Configuration option types. An option in KConfig is one of `bool`, `tristate`, or `constant`. A `bool` option means the code will either be *statically* compiled into the kernel binary or excluded, whereas `tristate` allows the code to be compiled as a Loadable Kernel Module (LKM)—a standalone object that can be loaded at runtime. A `constant` option can have a string or numeric value that is provided as a variable to the kernel code. An option can depend on another. KConfig employs a recursive process to enforce configuration dependencies[14] by recursively selecting dependent options and deselecting options whose dependencies are not met. The final kernel configuration has valid dependencies but could be different from user input (for instance, if the input configuration violates dependency requirements).

2.3. Configuration templates

The Linux kernel ships with a number of *manually crafted* configuration templates. But, configuration templates are not a solution to the kernel debloating challenges due to their hardcoded nature and need for manual intervention—they cannot adapt to different hardware platforms and do not have knowledge of application requirements. For example, a kernel built from `tinyconfig` cannot boot on standard hardware,[20] not to mention support any other applications. Some tools[12,13] treat `localmodconfig` as a minimized configuration. But, `localmodconfig` shares the same set of limitations of static configuration templates: it does not debloat configuration options that control statement-or function-level C preprocessors and does not deal with loadable kernel modules.

The templates `kvmconfig` and `xenconfig` are customized for kernels running on KVM and Xen setups. They provide domain knowledge of the underlying virtualization and hardware environment. In Section 6.5, we will use the information encoded in these templates to explore domain-specific debloating.

Figure 1. Kernel configuration that controls different granularities of kernel code in Linux kernels. The patterns are common in other kernels such as FreeBSD, L4, and eCos.

```
static int send_signal(...) {
  int from_ancestor_ns = 0;
#ifdef CONFIG_PID_NS
  from_ancestor_ns = ...;
#endif
  return __send_signal(...,
         from_ancestor_ns);
}
```

(a) Statement (C preprocessor)

```
#ifdef CONFIG_TRANSPARENT_HUGEPAGE
struct page *vm_normal_page_pmd(...) {
  unsigned long pfn = pmd_pfn(pmd);
  ...
out:
  return pfn_to_page(pfn);
}
#endif
```

(b) Function (C preprocessor)

```
cachefiles-y := bind.o daemon.o ...

cachefiles-$(CONFIG_CACHEFILES_HISTOGRAM)\
        += proc.o

obj-$(CONFIG_CACHEFILES) := cachefiles.o
```

(c) Object file (Makefiles)

2.4. Linux kernels in the cloud

Linux is still the dominant OS kernel in virtual machines (VMs) provided by cloud services. We focus on Ubuntu and Amazon Linux 2 as the representative kernels in the cloud. We analyze the Linux kernels provided by vendor VM images. We find that the cloud vendors all debloat the vanilla Linux kernel to some extent. However, the debloating efforts are manual and sometimes ad hoc. As shown in Table 1, the customization by cloud vendors is often done by directly removing loadable kernel modules (LKMs). Table 1 shows the inconsistency between the actual number of LKM files and the number of LKMs recorded at the compilation time during kernel builds. It shows that the kernels are initially built with many more LKMs that are removed later. A drawback of manually trimming LKM binaries is the potential for violating dependencies (an LKM could depend on multiple other LKMs that may have been manually trimmed).

Importantly, there is significant potential for improvements, that is, further debloating the kernel based on application requirements. The bottom half of Table 1 compares the kernels provided by cloud vendors against the debloated variants produced by Cozart for the official Ubuntu and Amazon Linux 2 (the baselines). We use the Apache Web Server (HTTPD) as the target application. Overall, Cozart can achieve kernel code reductions by 34.91% for Ubuntu and up to 40.72% for Amazon Linux 2.

We also study the minimized kernel used by Amazon FireCracker, a micro VM specialized for Function as a Service (AWS Lambda). The Firecracker kernel is based on Linux with handcrafted kernel configuration (910 configuration options). Table 1 shows that kernel debloating can further reduce such a minimized kernel by 17.69% using HTTPD as the target application.

3. COZART

Cozart is a new OS kernel debloating framework. It integrates our solutions to deal with the accidental limitations in its design and to support for analyzing the essential limitations. Cozart shares high-level principle with the state-of-the-art kernel debloating techniques—it traces the kernel footprint of (target) application workloads to determine the required kernel. The debloated kernel will only include those kernel features instead of all available ones or those enabled in default configurations. Cozart distinguishes itself from the other state-of-the-art kernel debloating techniques[6, 16–18, 21, 25] in the following aspects:

Table 1. Configuration options and sizes of the Linux kernels provided by Ubuntu, Google, AWS, and Azure.

	Kernel	# Options (bin/LKM)	Size
Ubuntu	Ubuntu 18.10 Cloud	2495/5147	62007918
Debloated by COZART		–51.28%/–99.86%	–65.09%
AWS	Amazon Linux 2	1214/946 (929)	58917847
Debloated by COZART		–29.82%/–97.46 (–97.42)%	–59.28%
AWS	Firecracker	910/0	15136127
Debloated by COZART		–17.69%/NA	–19.61%
Google	Ubuntu 18.10 Minimal	2454/989 (4993)	57155890
AWS	Ubuntu 18.10 Minimal	1966/949 (3075)	53605858
Azure	Ubuntu 18.10	1745/859 (1799)	53312392

"Size" includes kernel binary and LKMs.

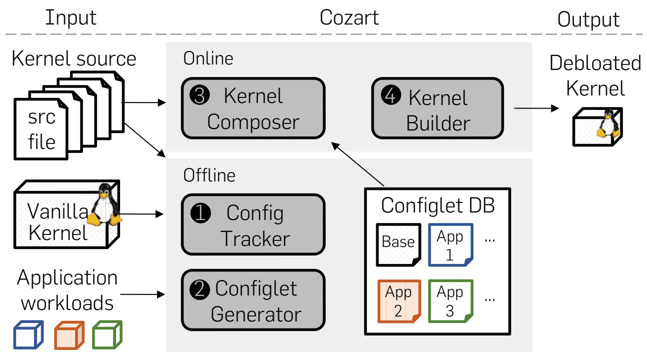
- **End-to-end visibility.** Cozart leverages the visibility of the hypervisor to implement end-to-end observations that can trace both the kernel boot phase and application workloads. We build Cozart on top of QEMU. Cozart targets OS kernels of cloud VMs.
- **Composability.** A key principle of Cozart is to make kernel configuration *composable*, by dividing them into a set of Configlets. A Configlet is a set of configuration options that is either (a) used to boot the kernel on a given deployment environment (e.g., a VM) or (b) ones needed by a target application (e.g., HTTPD). The former are named Baselets and the latter, Applets. A Baselet is not necessarily the minimal set of configurations needed to boot on a specific hardware but rather a set of configuration options that Cozart traces during the boot phase. A Baselet can be composed together with one or more Applets to generate the final kernel configuration.
- **Reusability.** Configlets can be stored in databases and be reused as long as the deployment environment and the application binaries do not change. Cozart moves kernel tracing and application workload runs off the critical path of the kernel debloating. This reusable nature avoids repetitive tracing and workload runs, making Configlet generation a one-time effort.
- **Support for fast application deployment.** Reusability and composability support fast deployment. Given a deployment environment and the target application(s), Cozart can effectively retrieve the Baselet and Applets and compose them into the required kernel configuration and then build the debloated kernel using the generated configuration.
- **Fine-grained configuration tracing.** Cozart uses program counter-based tracing provided by QEMU to identify configuration options based on low-level code patterns as described in Figure 1.

Figure 2 presents the architecture of Cozart. It requires three types of inputs: (1) the kernel source code, (2) the baseline configuration, and (3) the application workloads that drive the kernel execution.

Cozart operates in the following two phases (see Figure 2):

- **Offline phase.** Config Tracker is used to track configuration options required by the deployment environment and the target application ❶; Configlet Generator generates Configlets (such as both Baselets and Applets) and stores them in Configlet DB ❷.
- **Online phase.** Given the target deployment environment and applications, Cozart uses the Kernel Composer to generate the configuration by composing the corresponding Baselet and Applets ❸ and then uses Kernel

Figure 2. Cozart overview: Config tracker records configuration options used by the target application. Configlet generator processes these options into Configlets and stores them in Configlet DB. Kernel composer produces the final configuration. Kernel builder builds the debloated kernel.



Builder to build the debloated kernel ❹.

3.1. Tracking configuration options

Cozart tracks configuration options during the kernel execution driven by the target application. Cozart is expected to trace applications individually to generate Applets. It can later compose multiple Applets online. Configuration tracking is done via the following three steps.

Tracing Program Counters (PCs). Cozart uses the PC register to capture the addresses of instructions that are being executed. Our implementation uses the `exec_tb_block` provided by QEMU. To ensure that the traced PCs belong to target applications instead of other processes (e.g., background services), Cozart uses a *customized* `init` script that does not start any other applications. This script mounts filesystems (`/tmp`, `/proc`, and `/sys`), enables network interfaces (`lo` and `eth0`) and, finally, starts the application directly after kernel boot.

We disable Kernel Address Space Layout Randomization (KASLR) to be able to correctly map the addresses to source code. Note that this is only done during the tracking phase—*the debloated kernel can still use KASLR*.

Mapping PCs to source code statements. Cozart uses `addr2line` to map a PC to the corresponding statement in the source code to identify the kernel modules used at runtime. `addr2line` uses the debugging information in the kernel binary.

Loadable kernel modules (LKMs) need additional handling. An LKM is a standalone binary that can be loaded into a kernel at runtime. Because LKMs are not loaded into memory at a fixed address, `addr2line` cannot be directly applied. In Cozart, we use `/proc/modules` to obtain the start address of each of the loaded LKMs and then map the PCs to statements in the LKM binary. An alternative is to leverage `localmodconfig`, a kernel utility that outputs currently loaded LKMs. However, `localmodconfig` only provides information at the module granularity and cannot help debloat fine-grained intra-LKM configuration options.

Attributing statements to configurations. Cozart attributes source code statements to configuration options based on the three patterns presented in Figure 1. For the C preprocessor based patterns (see Figure 1a and b), Cozart analyzes C source files to extract the preprocessor directives (e.g., `#ifdef`) and then checks whether statements within these directives are executed. For Makefile-based patterns (see Figure 1c), Cozart determines if a configuration option should be selected at the granularity of the object files. For instance, in Figure 1c, `CONFIG_CACHEFILES` needs to be selected if any of the corresponding files (`bind.o`, `cachefiles.o`, or `daemon.o`) are used.

3.2. Generating CONFIGLETS

Baselets and Applets are generated during the (offline) phase in which configuration options are tracked. Cozart marks the end of the boot phase using a *landmark program*—a C program created by us that uses `mmap` to map an empty stub function to a pre-defined "magic address" (we use 0x333333333000 and 0x222222222000 for the start and the end respectively) and invokes the stub function. The `init` script described in Subsection *Tracing Program Counters* calls the landmark before running the target application. Therefore, Cozart can recognize the (end of the) boot phase based on the magic address in the PC trace.

Cozart takes the configuration options from the application and filters out the hardware-related options that are observed during the boot phase. These hardware features are defined based on their location in the kernel source code (e.g., options located in `/arch`, `/drivers`, and `/sound`). We do not exclude the possibility that a hardware-related option may only be observed during the execution of an application, for example, it loads a new device driver on demand (in such cases, the application is hardware-specific).

3.3. Composing CONFIGLETS

Cozart compOSs the Baselet with one or more Applets to produce a final configuration that is used to build the kernel. Cozart first creates a union of the options in all the Configlets into an initial configuration and then resolves the dependencies between them using an SAT solver. It models the configuration dependencies as a Boolean satisfiability problem and uses a SAT solver to generate a final configuration—a valid configuration is one that satisfies all the specified dependencies between configuration options. This is needed because KConfig does not ensure all the selected options are included but deselects unsatisfied dependencies. We follow the SAT-based modeling for kernel configuration described in literature[7] and use PicoSAT in Cozart.[5] It is possible that the SAT solver returns multiple valid solutions. In this case, our implementation uses the first solution returned from the solver.

3.4. Kernel builder

Cozart uses the standard kernel build system (KBuild for Linux) to build the debloated kernel based on the configuration composed using the Configlets. As building is on the critical path of kernel debloating, Cozart optimizes the build time by leveraging the incremental build support of modern tools (e.g., `make`). Cozart caches the previous build results (e.g., the object files and LKMs) to avoid redundant compilation and linking. When a configuration change happens, only the modules with the changes to internal configuration options need to be rebuilt whereas the other files can be reused. We find this feature to be particularly useful when a debloated kernel needs to support a new application because most applications share many modules but only differ in a small number of modules.

4. STUDY METHODOLOGY

We use six popular open source server applications that are commonly deployed in cloud environments, viz. HTTPD (v2.4), NGINX (v1.15), Memcached (v1.5), MySQL (v5.7), PHP (v7.2), and Redis (v4.0). For each application, we use both its official test suite as well as performance benchmarks to drive the kernel, based on which we use Cozart to generate the Configlets.

We use the official benchmarks to measure the application performance running on the debloated kernel and to verify that no regression was introduced by the debloating. All the performance experiments were performed on a KVM-based VM with 4 VCPUs and 8GB RAM running on an Intel Xeon Silver 4110 CPU operating at 2.10 GHz. We report the benchmark results as the average of 20 runs.

Our baseline kernel is Linux kernel v4.18.0 from the latest Ubuntu 18.10 Cloud Image. We also repeated all our experiments on Linux kernel v5.1.9 from Fedora 30 and observed similar results. Therefore, we only present the results from the Ubuntu 18.10 Cloud Image as the baseline.

5. DEBLOATING EFFECTIVENESS

We use Cozart to debloat the baseline kernel for our target applications. Cozart achieves more than 80% kernel reduction (see Table 3). We verify that all the debloated kernels can not only boot but run application workloads successfully, such as the performance benchmarks and test suites.

Boot time reduction. The second column in Table 2 shows the boot time (and the reduction with regards to the baseline kernel) of the debloated kernel for each application. We measure the boot time by reading from `/proc/uptime` that records the duration of the system being on since its last restart. The kernels debloated by Cozart achieve a time reduction of close to 14%. The reduction is attributed to the savings in loading smaller kernel images and the skipping of initialization and device registration for the removed modules (e.g., Fuse filesystem and Hot Plug PCI controller).

Table 2. Boot time, application performance and memory reduction for debloated kernels.

Application	Boot Time (ms)	App. Perf. (%)	Mem. Save(KB)
Baseline	646	0.00	0
Apache HTTPD	561 (–13.16%)	+1.70%	+4,012
Memcached	557 (–13.78%)	+3.44%	+4,012
MySQL	558 (–13.62%)	+0.29%	+4,016
Nginx	562 (–13.00%)	+3.53%	+4,016
PHP	556 (–13.93%)	+0.21%	+4,012
Redis	556 (–13.93%)	+3.49/3.78%	+4,012

Application performance. We benchmark the performance of applications running on debloated kernels generated by Cozart and compare the results to the baseline. We do not expect any performance improvements. We expect no performance regressions either as Cozart does not remove performance optimization code.

Table 2 (the third column) shows the improvements of application performance running on debloated kernels with regards to the baseline. There is no performance regression; instead, the debloated kernels show marginal performance improvements (for all the applications) mainly because applications load faster and have smaller memory overheads.

Memory savings. We define memory savings as the reduction in the memory region needed to host the loaded kernel binary, measured by `MemTotal` as read from `/proc/meminfo`. The numbers in Table 2 are aligned to the page size. We observe about 4MB memory savings across the debloated kernels for single applications. Applications have similar reduction results (discussed in Section 6.1).

> Summary: The debloated kernels produced by Cozart are all stable—they can directly boot and support the expected application workloads. They can boot 13+% faster than baseline kernels and achieve more than 4MB memory savings at runtime.

6. FINDINGS AND IMPLICATIONS

This section selectively presents our findings. Please refer to the original paper for the complete discussion.

6.1. Boot phase visibility

Experiences with the state-of-the-art tools. Existing tools do not have automatic solutions to tackle the boot phase because they use `ftrace` for kernel tracing, which can only be started after the boot process is complete.

To work around this limitation, Undertaker[16] turns on `ftrace` as early as possible, that is, at the initialization of the RAM disk (`initrd`). However, we find that it is still not early enough, because hardware detection happens before mounting RAM disk. As a result, some disk drivers (e.g., the SCSI disk support) are not captured by Undertaker, and even the disk support itself has to be whitelisted. To make the debloated kernel bootable, Undertaker eventually requires users to set a small whitelist to ensure certain configuration options are included. However, as articulated in recent literature,[6] "that brings the user back to the original configuration issue."

LKTF[13] attempts to address the invisibility of the boot phase using a search-based approach. It fills in configuration options into the unbootable kernel generated by Undertaker until the kernel eventually boots. LKTF is supposed to take approximately 5 h to generate a bootable kernel.[6] We were not able to run LKTF because its implementation is hard-coded to a few old kernel versions. Also, the debloated kernel generated by LKTF is not guaranteed to include all the original security and performance configurations.

Boot-phase visibility with COZART. Cozart addresses the visibility limitation using a lower level of tracing—at the hypervisor level. This brings a key advantage of Cozart: *every debloated kernel by* Cozart *can boot and run stably.*

Cozart generates the Baselet for Ubuntu 18.10, which contains 1212 (out of 2443) statically-linked modules and 7 (out of 5001) LKMs. We reduce 1231 static-linked modules and 4994 LKMs compared with the baseline kernel (Ubuntu 18.10). Take drivers as an example. The vanilla kernel aims to support a variety of hardware and thus contains drivers for different devices. Cozart only includes the one observed in the traces (i.e., those used by the applications), such as `acpi`, `scsi`, `cpufreq`, `tty`, `char`, and `dma` drivers.

Summary: Boot-phase visibility is critical to producing a bootable kernel. Cozart leverages the tracing feature provided by the hypervisor to support end-to-end observability (such as the complete boot phase and application workloads) and produce stable kernels that can always boot and run applications stably.

6.2. Composability

We evaluate Cozart's composability with the following three cases: (1) *individual application:* for each application, we use Cozart to compose an Applet and the Baselet to generate a tailored debloated kernel; (2) *individual application in a container:* for each application, we use Cozart to compose an Applet along with the Configlet of the Docker Runtime (`dockerd`) in conjunction with the Baselet; (3) *multiple applications:* we use Cozart to compose a debloated kernel to support a LAMP stack. We use a separate application, `phpMyAdmin` (a MySQL administration tool with PHP interfaces), to validate the LAMP stack as well as individual application works.

Table 3 shows the debloating results of the first two types of compositions, in comparison with the baseline kernel. The reason why all the reduction numbers look similar is that 96%–99% of the configuration options are drawn from the Baselet that is generated from the same VM. Cozart compOSs the Baselet with the target Applet to achieve 17+% and 86+% size reductions for static-linked kernel binaries and LKMs, respectively.

We next use Cozart to generate the Configlet for *Docker Runtime* (`dockerd`) using the standard `hello-world` example that starts `dockerd`, pulls the `hello-world` image, and launches a container. We generate debloated kernels by composing multiple Applets (such as the target application and `dockerd`) and the Baselets. The lower half of Table 3 shows the reduction achieved by debloated Linux kernels running applications in Docker containers. We find that the `dockerd` Configlet contains most of the Applets already. The debloated kernels have similar reduction ratios.

For the LAMP stack, we use Cozart to compose the Configlets of each application (HTTPD, MySQL, and PHP), together with the Baselet to generate a debloated kernel for the entire stack. We deploy `phpMyAdmin` and ensure that the functionalities of all components work properly. The debloated kernel shows a size reduction of 17.13% for the statically linked kernel binary and 86.80% for LKMs.

Summary: The composability of Cozart enables users to maintain application-specific kernel configuration in Applets and deployment-specific kernel configuration in Baselets. A complete kernel configuration can be generated by composing the Applets and Baselet.

6.3. Fast application deployment

A limitation of existing techniques, when one wants to deploy a new application onto a debloated kernel (i.e., customized for a different application), is that the entire process from tracing to the recompilation must be repeated. Thus, fast application deployment cannot be supported.

We argue that the aforementioned limitation is caused by the design of existing techniques that make tracing an integral component of the debloating process—that is, the tool has to first trace the kernel by running application workloads and then compile the kernel based on the results. Our work shows that tracing requires significantly more time than

Table 3. Characteristics of the debloated kernel generated by Cozart based on Configlet composition.

	# Remaining kernel modules		Kernel size reduction			
Application	**Static**	**LKM**	**Static**	**LKM**	**Default**	**Overall**
Ubuntu Vanilla (Baseline)	2443	5001	0%	0%	0%	0%
Base configlet	1212	7	−17.21%	−86.80%	−29.52%	−83.53%
Apache HTTPD	1213	7	−17.19%	−86.80%	−29.50%	−83.52%
Memcached	1215	7	−17.19%	−86.80%	−29.50%	−83.52%
MySQL	1221	7	−17.13%	−86.80%	−29.46%	−83.51%
NGINX	1215	7	−17.19%	−86.80%	−29.50%	−83.52%
PHP	1216	7	−17.21%	−86.80%	−29.50%	−83.53%
Redis	1217	7	−17.17%	−86.80%	−29.49%	−83.52%
Docker	1232	35	−17.02%	−75.11%	−27.30%	−83.01%
Docker + Apache HTTPD	1233	35	−16.99%	−75.10%	−27.27%	−83.00%
Docker + Memcached	1233	35	−16.99%	−75.10%	−27.27%	−83.00%
Docker + MySQL	1239	35	−16.93%	−75.10%	−27.22%	−82.99%
Docker + NGINX	1233	35	−16.99%	−75.10%	−27.27%	−83.00%
Docker + PHP	1236	35	−16.98%	−75.11%	−27.30%	−83.01%
Docker + Redis	1237	35	−16.98%	−75.10%	−27.26%	−83.00%

"Default" refers to a conservative baseline that only includes the LKMs that are autoloaded based on Ubuntu 18.10 configuration (Ubuntu includes all the LKMs but does not load every one by default). "Overall" refers to the overall kernel space (all modules).

compiling. Specifically, the tracing time depends on the application workloads (test suites or benchmarks)—a more complete workload would lead to higher tracing overhead. With Cozart, tracing can be decoupled from the compilation and, thus, can be done offline, as long as the resulting Configlets are preserved. *If we have the* Configlets *for the target applications, debloating a kernel takes less than 2 min when compiled from scratch.*

We also explore *incremental builds from previously debloated kernels*, where we add a *new* application on top of an existing debloated kernel. Such incremental builds are effective, attributed to the fact that many applications share common configuration options and only slightly differ. As a result, Cozart only needs to (re-)compile these small number of additional modules. It reuses the majority of the previously built binaries. It only takes tens of seconds (in the presence of cold caches) to rebuild the kernel for supporting a new application and only a few seconds if the cache is hot (typically when dedicated build servers are in use).

> Summary: Kernel debloating can be done within tens of seconds if the configuration options of target applications are known. Cozart's composability allows Applets to be prepared offline. In fact, deploying a new application can also be done on a debloated kernel within tens of seconds.

6.4. Coverage

One essential concern of applying existing techniques is the *completeness* of the debloated kernel. If the target application reaches code that is not included in the debloated version, the kernel could potentially crash or result in errors.

The prior studies on debloating techniques implicitly assume that running a benchmark could cover the kernel footprint of the target applications. Take Web servers as an example. Prior studies use simple benchmarks that access static documents to generate debloated kernels.[16]

Table 4 shows the coverage (in terms of number of statements) of each target application when running the official performance benchmark suites, as measured by `gcov`. The coverage reached by the official benchmark is low, ranging from 6% to 26%. Typically, the benchmarks only exercise the steady states of the target applications but miss the other part of the application program (e.g., other features, error handling, and recovery procedures).

Test suites can, potentially, complement benchmarks; they are designed to exercise different parts of the application software. As shown in Table 4, test suites provide higher code coverage when compared to the benchmarks. However, the code coverage of the test suites is also imperfect—the coverage varies from 29% to 73%.[a] We conclude that the lack of high-coverage tests is an essential limitation of dynamic-tracing based debloating techniques. Automated testing techniques such as fuzz testing and concolic testing might improve the coverage and alleviate the limitations. On the other hand, they are not silver bullets. Note that code coverage is not equal to kernel reach. A test suite with 100% coverage can cover all possible system calls invoked by the application, but not all possible kernel states.

Table 4. Statement coverage and number of configuration options of the target applications reached by the official performance benchmarks versus official test suites.

	Test suite		Benchmark	
Application	Coverage	# Options	Coverage	# Options
HTTPD	29%	76	13%	97
Memcached	73%	120	26%	80
NGINX	69%	123	8%	80
MySQL	68%	120	13%	93
PHP	61%	115	6%	35
Redis	57%	115	11%	65

The configuration options discovered by benchmarks are not always included in the Applets from test suites. For example, in Memcached, `CONFIG_PROC_SYSCTL` only exists in Applets generated by the benchmark when Memcached reads the maximum number of file descriptors to check whether it can allocate more connections in an overloaded situation.

> Summary: An essential limitation of dynamic-tracing-based debloating techniques is the imperfection in test suites and benchmarks. Specifically, the official test suites of many open source applications have low code coverage.

6.5. Using domain-specific information

Another essential limitation of existing kernel debloating techniques is the inability to distinguish between kernel code that is *executed* during the tracing versus code that is *needed*. It is possible that the code is not needed but happens to be on the execution paths. For example, in VMs, kernel code for BIOS, thermal control, and power management is executed but not needed to boot the VM.

We also studied the opportunities to further debloat the kernel, based on domain knowledge, to remove the code. We study both, KVM-based VMs and Xen VMs. We leverage configuration templates, `kvmconfig` and `xenconfig`, in the Linux source tree as domain-specific information. For example, `kvmconfig` uses `virtio` as the I/O interface for both net and block devices whereas `xenconfig` uses the `Xen` frontend (`Dom0`) as the main I/O interface; the original Baselet also uses other I/O interfaces (such as the SCSI disk driver) that are not needed for KVM or Xen VMs.

We use these two templates to replace the original Baselets to further debloat the already-debloated kernel for KVM and Xen. *We observe a 40+% and 39+% kernel size reduction of KVM and Xen, respectively, compared with the original* Baselet *and a result of 40+% and 41+% reduction in the number of configuration options for KVM and Xen, respectively.* The reduction comes from the fact that many kernel modules (e.g., partition types, processor features, BIOS, thermal control, power management, ACPI, and input devices) are executed by the baseline kernel at the boot time; however, those modules are not needed if the deployment is on KVM or Xen.

[a] It is reported, however, that test suites for industrial software have much higher code coverage.[10]

SUMMARY: Domain-specific information can be used to effectively remove the kernel code that is executed in the baseline kernel but is not needed at deployment time.

7. GENERALITY

We ported COZART to debloat the L4 microkernel[1, 8] and the AWS Firecracker kernel.[2, 3] The L4/Fiasco microkernel also uses KConfig as the configuration language as Linux does. For Firecracker, we modified the kernel bus from MMIO to PCI to enable QEMU tracing and changed the bus back before booting on Firecracker.

L4 microkernel. The result of applying COZART to L4[1] is surprising—the debloated kernel is 47% smaller, compared to the default. The reason for the significant reduction is that the default L4/Fiasco configuration enables a heavy kernel debugger, `CONFIG_JDB`, that contributes to a big part of the final kernel size. The debloated kernel no longer includes this module. Although it may sound straightforward that removing a debugger makes the kernel smaller in size, without an automatic tool such as COZART, it is nontrivial to examine and select every configuration option manually.

Firecracker. Although still based on Linux, Amazon Firecracker kernel is a special case as the kernel is extensively minimized to optimize for Function-as-a-Service workloads. We apply COZART on the Firecracker kernel using the target applications (Section 4) to understand the space for application-specific kernel debloating, especially for an extensively optimized kernel such as Firecracker.

Overall, COZART reduces the number of kernel configuration options from 910 to 729 (a 20% reduction), leading to a 19.76% kernel size reduction and an 11% faster boot time (reduced from 104 to 92.5 ∝s).

SUMMARY: Application-oriented kernel debloating can be used to further kernel code reduction even for microkernels and extensively customized kernels.

8. CONCLUSION

OS kernel debloating techniques have not received wide-ranging adoption due to the instabilities of the debloated kernels, the lack of speed, completeness issues, and the need for manual intervention. This paper studies such debloating techniques with the goal of making them practical in real-world deployments. We identify the limitations of existing techniques that hinder their practical adoption. We developed COZART, an automatic kernel debloating framework that enables us to conduct a number of experiments on different types of operating systems with a wide variety of applications. We share our solutions in addressing the accidental limitations of OS kernel debloating and we also discuss the challenges and opportunities in addressing the essential limitations toward practical debloating techniques. We have made COZART and the experiment data publicly available at: https://github.com/hckuo/Cozart.

Acknowledgments

Mohan is supported in part by the Office of Naval Research (ONR) grant N00014-17-S-B010. Xu is supported in part by NSF grants CCF-1816615, CCF-2029049, CNF-1956007, and a Facebook Distributed Systems Research award.

References

1. FIASCO: The L4Re microkernel. http://os.inf.tu-dresden.de/fiasco. Retrieved on October 2019.
2. Firecracker: Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/. Retrieved on October 2019.
3. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.-M. Firecracker: lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Santa Clara, CA, February 2020).
4. Alharthi, M., Hu, H., Moon, H., Kim, T. On the effectiveness of kernel debloating via compile-time configuration. In *Proceedings of the 1st Workshop on SoftwAre debLoating And Delayering* (Amsterdam, Netherlands, July 2018).
5. Biere, A. Picosat essentials. *J. Satisfiability, Boolean Modeling Comput. 4*, 2-4 (2008), 75–97.
6. Corbet, J. A different approach to kernel configuration, 2016. https://lwn.net/Articles/733405/.
7. Dietrich, C., Tartler, R., Schröder-Preikschat, W., Lohmann, D. A robust approach for variability extraction from the linux build system. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)* (Salvador, Brazil, September 2012).
8. Elphinstone, K., Heiser, G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Farmington, PA, November 2013).
9. Hubaux, A., Xiong, Y., Czarnecki, K. A user survey of configuration challenges in Linux and eCos. In *Proceedings of 6th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'12)* (Leipzig, Germany, January 2012).
10. Ivanković, M., Petrović, G., Just, R., Fraser, G. Code coverage at Google. In *Proceedings of the 2019 12th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2019)* (Tallinn, Estonia, 2019).
11. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R.A., Stoica, I., Patterson, D.A. Cloud programming simplified: A Berkeley view on serverless computing. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley, 2019.
12. Kang, J. A practical approach of tailoring Linux kernel. In *The Linux Foundation Open Source Summit North America* (Los Angeles, CA, September 2017).
13. Kang, J. An empirical study of an advanced kernel tailoring framework. In *The Linux Foundation Open Source Summit* (Vancouver, BC, Canada, August 2018).
14. kernel.org. Kconfig, 2018. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.
15. Kuo, H., Gunasekaran, A., Jang, Y., Mohan, S., Bobba, R.B., Lie, D., Walker, J. MultiK: A framework for orchestrating multiple specialized kernels. arXiv:1903.06889 (2019).
16. Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., Kapitza, R. Attack surface metrics and automated compile-time os kernel tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)* (San Diego, CA, USA, February 2013).
17. Lee, C.-T., Lin, J.-M., Hong, Z.-W., Lee, W.-T. An application-oriented Linux kernel customization for embedded systems. *J. Inf. Sci. Eng. 20*, 6 (2004), 1093–1107.
18. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Shanghai, China, October 2017).
19. Passos, L., Queiroz, R., Mukelabai, M., Berger, T., Apel, S., Czarnecki, K., Padilla, J. A study of feature scattering in the Linux kernel. *IEEE Trans. Software Eng. (TSE) 47*, 1 (2021), 146–164.
20. Pitre, N. Shrinking the kernel with an axe, 2018. https://lwn.net/Articles/746780/.
21. Stengel, K., Schmaus, F., Kapitza, R. EsseOS: Haskell-based tailored services for the cloud. In *Proceedings of the 12th International Workshop on Adaptive and Reective Middleware (ARM'13)* (Beijing, China, December 2013).
22. Tsai, C.-C., Jain, B., Abdul, N.A., Porter, D.E. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (London, UK, April 2016).
23. Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R. Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, August 2015).
24. Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., Pasupathy, S. Do not blame users for misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Farmington, PA, November 2013).
25. Youseff, L.M., Wolski, R., Krintz, C. Linux kernel specialization for scientific application performance. Technical Report 2005-29. University of California Santa Barbara, 2005. https://www.cs.ucsb.edu/research/tech-reports/2005-29.

Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu ([hckuo2, jianyan2, sibin, tyxu]@illinois.edu), University of Illinois, Urbana-Champaign, USA.

Copyright held by owner/author.