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

HSUAN-CHI KUO, University of Illinois at Urbana-Champaign
JIANYAN CHEN, University of Illinois at Urbana-Champaign
SIBIN MOHAN, University of Illinois at Urbana-Champaign
TIANYIN XU, University of Illinois at Urbana-Champaign

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

ACM Reference Format:
Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article pomacs9 (March 2020), 27 pages. https://doi.org/10.1145/XXXXXXX

1 INTRODUCTION

1.1 Motivation

Commodity operating systems (OSes), such as Linux and FreeBSD, have grown in complexity and size over the years. However, each application usually requires a small subset of OS features [56, 74], rendering the OS kernel bloated. The bloated OS kernel leads to increased attack surface, prolonged boot time and increased memory usage. Application-oriented OS kernel debloating—*reducing the kernel code that is not needed by the target applications*—is reported to be effective in mitigating many of the above issues. For example, Kurmus *et al.* [46] report that 50%–85% of the attack surface can be reduced by debloating the Linux kernel for server software. Alharthi *et al.* [17] show that 34%–74% of known security vulnerabilities can be nullified in the Linux kernel if it only includes kernel modules that are needed by the target application(s).

Authors' addresses: Hsuan-Chi Kuo, hckuo2@illinois.edu, University of Illinois at Urbana-Champaign; Jianyan Chen, jianyan2@illinois.edu, University of Illinois at Urbana-Champaign; Sibin Mohan, sibin@illinois.edu, University of Illinois at Urbana-Champaign; Tianyin Xu, tyxu@illinois.edu, University of Illinois at Urbana-Champaign.

2476-1249/2020/3-ARTpomacs9 $15.00
https://doi.org/10.1145/XXXXXXX

Recent trends in Function-as-a-Service (FaaS) and microservices, where numerous kernels are often packed and run in virtual machines (VMs), further highlight the importance of kernel debloating. In these scenarios, most VMs run small applications [20, 21, 49, 52], and each application tends to be "micro" with a small kernel footprint [31, 37, 64]. Some of the recent virtualization systems, such as LightVM [52], require users to provide *minimalistic* Linux kernels specialized for target applications.

Debloating OS kernels by hand-picking kernel features is impractical. The complexity of commodity-off-the-shelf (COTS) OSes (e.g., Linux) makes manual debloating infeasible [16, 33, 46]. Linux 4.14, for example, has more than 14,000 configuration options with hundreds of new configuration options introduced every year (see Figure 2). Kernel configurators (such as KConfig [42]) do not help debloat the kernel but only provide a user interface for selecting configuration options. Given the poor usability and incomplete and even incorrect documentation [33], it is prohibitively difficult for users to select the minimal kernel configuration manually.

Recently, several *automated* kernel debloating techniques and tools have been proposed and built [25, 38, 39, 45–48, 52, 69, 71, 79]. Despite their different implementations, existing kernel debloating techniques share the same working principle with the following three steps: (1) running the target application workloads, meanwhile tracing the kernel code that were executed during the application runs; (2) analyzing the traces and determining the kernel code needed by the target applications; (3) building a debloated kernel that only includes the code required by the applications.

Configuration-driven (also known as feature-driven) kernel debloating techniques are the *de facto* method for OS kernel debloating [25, 38, 39, 46–48, 52, 69, 71, 79]. Most existing kernel debloating tools use configuration-driven 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 is corresponding to a kernel feature. A debloated kernel *only* includes features for supporting the target application workloads. As shown by recent studies, configuration-driven kernel debloating can effectively reduce the size of the kernel by 50%–85% [46], attack surface by 50%—85% [46], and security vulnerabilities by 34%–74% [17]. This paper focuses on configuration-driven kernel debloating and their open-source implementations.

However, despite their attractive benefits regarding security and performance, automated kernel debloating techniques are seldom adopted in practice, especially in production systems. This is certainly not due to a lack of demand—we observe that many cloud vendors (e.g., Amazon AWS, Microsoft Azure and Google Cloud) reduce the Linux kernel code in a handcrafted manner, which is not as effective as automated kernel debloating techniques (Section 3). The goal of this paper is to understand the limitations of kernel debloating techniques that hinder their practical adoption and to share our experience to shed light on addressing the limitations.

1.2 Accidental and Essential Limitations

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

- *No visibility of kernel boot phases.* Existing debloating techniques use `ftrace` that can only be initiated after the kernel boot and hence cannot observe what kernel code is loaded during the boot phase[1]. As a consequence, critical modules (*e.g.,* disk drivers) might be missing, and the debloated kernel generated by existing techniques oftentimes cannot boot [25]. Our experiments show that up to 79% of kernel features can only be captured by observing the boot phase. Even if the debloated kernel can boot, the kernel could miss certain performance and security features loaded at the boot time, leading to reduced performance and security.

[1] We show `ftrace`'s incapability of observing the boot phase even with the RAM disk based solution (`initrd`) in Section 7.1.

We find that many performance and security features are only loaded at the boot time, e.g., `CONFIG_SCHED_MC` for multicore support and `CONFIG_SECURITY_NETWORK`.

- *Lack of support for fast application deployment.* Using existing tools, debloating for a new combination of applications or deploying a new application on a debloated kernel would require going through the entire three-step debloating process of tracing, analysis, and building as described in Section 1.1. The process is time-consuming (can take hours or even days) and thus prevent agile application deployment—starting up a new application would have to wait until the generation of the debloated kernel.
- *Coarse-grained tracing.* Existing techniques use `ftrace`, which can only trace kernel code at the function level. However, function-level tracing is too coarse a granularity to track configuration options that affect intra-function code.
- *Difficulties in covering the complete kernel footprint.* Since kernel debloating uses dynamic tracing, it requires application workloads (typically encoded in benchmarks) to drive kernel code execution. However, it is challenging for benchmarks to cover the complete kernel footprint of the target applications, and the debloated kernel would crash if the application reaches kernel code that was not observed during tracing.
- *Not distinguishing what is executed versus what is needed.* Existing techniques treat any kernel features that are reached during the application workloads, even though the executed code may not be actually needed. For example, a second filesystem might be initialized but never needed.

To analyze the above limitations, we divide them into *essence* (the 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). We believe that the first three limitations are accidental and can be addressed by improving the design and implementation of the debloating techniques, while the last two limitations are essential that require efforts beyond the debloating techniques themselves.

1.3 Our Study and Results

This paper studies the limitations of existing OS kernel debloating techniques towards making OS kernel debloating practical, effective and fast. We focus on OS kernels deployed in cloud environments that are commonly used as guest OSes to run untrusted applications often with stringent performance requirements.

To study the limitations, we carefully designed and implemented an advanced OS kernel debloating framework named Cozart[2] built on top of QEMU [8]. Cozart employs a low-level tracing provided by QEMU to obtain visibility of the boot phase. Cozart uses instruction-level tracing to track the kernel code and map the kernel code to kernel configuration options. Cozart moves kernel tracing off the critical path: with Cozart, one can generate Configlets (a set of configuration options) specific to an application or a deployment environment *offline*. 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 previous build 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 (including Linux, the L4 microkernel and Amazon Firecracker) with a wide variety of

[2] Cozart is publicly available at https://github.com/hckuo/Cozart.

applications (including HTTPD, Memcached, MySQL, NGINX, PHP and Redis). Our studies lead to the following findings and results:

- Existing kernel debloating techniques initialize in-kernel tracing methods (*e.g.*, ftrace) too late and cannot observe the boot phase, which is critical to producing a bootable kernel. Cozart leverages the tracing feature provided by the hypervisor to support end-to-end observability (including the complete boot phase and application workloads) and produce stable kernels that can always boot and run applications stably. The debloated kernel by Cozart does not have performance regressions (but slightly improves application performance) while achieving more than 13% reduction of boot time and more than 4MB memory savings.
- Kernel debloating can be done within tens of seconds if the configuration options of target applications are known. The composability of Cozart allows users to prepare Applets offline. To deploy a new application on a debloated kernel, Cozart composes the Applet of the target application with the Configlets of the current kernel to generate a new debloated kernel within tens of seconds.
- Using instruction-level tracing (instead of `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, given that those can be done offline.
- An essential limitation of using dynamic-tracing based techniques for kernel debloating (which are the *de facto* approach) is the imperfect 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 the 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 by the 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 provided by the Linux kernel source.
- Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and extensively customized kernels (e.g., the Firecracker kernel). The reduction is significant: 47.0% for the L4 microkernel and 19.76% for the Firecracker kernel.

2 KERNEL CONFIGURATION

This section provides an overview of kernel configuration ecosystem using Linux as an example. Kernel debloating is not specific to Linux, and Cozart is generic and portable to different OSes.

2.1 Configuration Options

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

C preprocessors select code blocks based on `#ifdef` or `#ifndef` directives — configuration options are used (as macros) to determine whether or not to include conditional groups in the compiled kernel. Figures 1a and 1b show examples of C preprocessors for two configuration options, with the granularity of statements and functions, 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`

```
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)

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

are both configuration options. The detailed characteristics of kernel configuration patterns have been studied before [26, 29, 54].

Note that statement-level configuration options (Figure 1a) cannot be identified by function-level tracing (*e.g.*, `ftrace`) used by existing kernel debloating tools [38, 39, 46, 71]. We find that 31% of C preprocessors in Linux 4.18 are options at the statement level.

The number of configuration options in modern monolithic OS kernels is increasing rapidly as a result of the rapid growth of the kernel code and features. Figure 2 shows the growth of kernel configuration options of Linux and FreeBSD on a yearly basis. Taking Linux as an example, the Linux kernel 5.0, 4.0 and 3.0 have 16,527, 14,406, 11,328 configuration options respectively.

2.2 Configuration Languages

Different OS kernels employ a variety of configuration languages to instruct the compiler on which code to include in the compiled kernel. For example, Linux and L4/Fiasco use KConfig [68], eCos uses a component definition language (CDL) [75], and FreeBSD uses a simple key-value format [12]. Despite different language syntaxes, the configuration language allows the definition of configuration options and dependencies between them. Without loss of generality, we use KConfig as an example of kernel configuration semantics through the rest of this paper.

Configuration Option Types. A configuration option in KConfig could be `bool`, `tristate` or `constant`. A `bool` option means the code will either be *statically* compiled into the kernel binary or excluded, while `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 variables to the kernel code (*e.g.*, kernel log buffer size: `CONFIG_LOG_BUF_SHIFT`). In Linux 4.18, 48% of configuration options are `bool` (must be compiled into the kernel,*e.g.*, threads and memory), 49% are `tristate` (mostly device drivers), and the remaining 3% is `constant`.

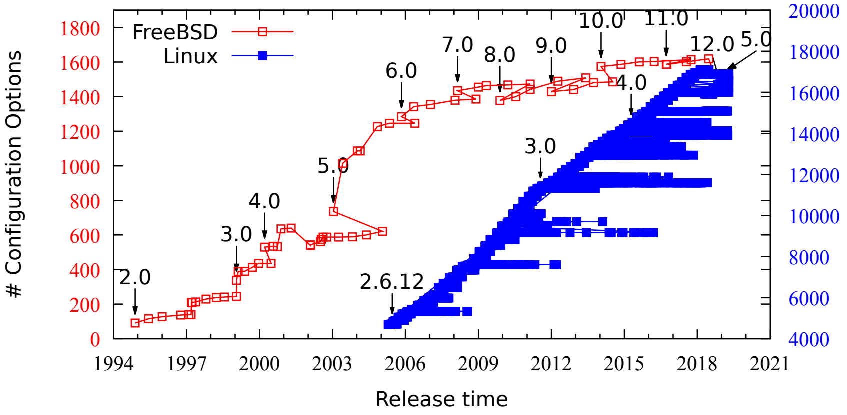

Fig. 2. The unrelenting growth of configuration options in Linux and FreeBSD showing the need for debloating. Each point is a release tag of the code repositories.

Configuration Dependencies. Two configuration options A and B can have one of the following dependencies:

- A depends on B: A can be selected only if B is selected (*e.g.*, TMPFS_XATTR depends on TMPFS);
- A selects B: B is forcefully selected if A is selected (*e.g.*, TMPFS_POSIX_ACL selects TMPFS_XATTR);
- A implies B: B is selected by default if A is selected (*e.g.*, FUTEX implies RT_MUTEXES). However, implied options (B in the above example) can be unselected explicitly.

The Linux kernel configuration system, KConfig, employs a recursive process to enforce the above configuration dependencies [42]. It recursively selects configuration options that are implied and selected and deselects options whose depend relationship is not met. The final kernel configuration always follows valid dependencies but could be different from user input (if the input configuration violates dependency requirements).

2.3 Configuration Templates

The Linux kernel ships with a number of *manually crafted* configuration templates that includes:

- defconfig for the default configuration,
- tinyconfig includes very basic modules [58]
- localmodconfig disables all loadable kernel modules that are not currently loaded.
- kvmconfig enables options for KVM guest support.
- xenconfig enables options for Xen Dom0 and guest support.

We would like to note that configuration templates are not a solution to the kernel debloating challenges because the configuration templates are all handcrafted and hardcoded. They cannot adapt to different hardware platforms and do not have knowledge of the application requirements. For example, we find that the kernel built from tinyconfig cannot boot on standard hardware [58], not to mention supporting applications. Some prior work [38–40] treats localmodconfig as a minimized configuration; however, 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.

`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 7.5, we use the information encoded in these templates to explore the potential of debloating kernels with domain knowledge.

3 LINUX KERNELS IN THE CLOUD

Since we target kernel debloating for cloud-based environments, we conduct a study on existing OS kernels provided by major cloud service providers, including Amazon AWS, Google Cloud and Microsoft Azure.

In current cloud services, Linux is the dominant OS kernel in VMs provided by Amazon AWS, Microsoft Azure and Google Cloud. Most cloud vendors only provide Linux and Windows servers with Ubuntu, being the most popular distribution [27, 35]. Amazon provides its customized Linux-based server named Amazon Linux 2 [1] for EC2 instances. Therefore, we focus on Ubuntu and Amazon Linux 2 as the representative kernels in the cloud.

We analyze the Linux kernels provided by official Virtual Machine (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 LKMs from the file system (or adding new 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. Our observations show that the kernels are initially built with many more LKMs that are then removed later from the built kernels. 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 a significant potential of improvements to further debloat the kernel based on the applications. Table 1 (the bottom half) compares the kernels provided by cloud vendors with the debloated variants produced by Cozart for the official Ubuntu and Amazon Linux 2, which are the baselines respectively. We use the Apache Web Server (HTTPD) as the target application.

As shown in Table 1, Cozart can further reduce the kernel size significantly on top of the manually customized Linux kernels. Overall, Cozart can achieve reductions of the kernels by

Table 1. Configuration options and sizes of the Linux kernels provided by Ubuntu, Google, AWS, and Azure. "Kernel size" includes kernel binary and LKMs. "# Options" is categorized into "kernel binary / LKM." Many kernels are customized by removing or adding LKMs; the actual LKMs are inconsistent with the number at the compilation time (in parentheses).

	Kernel	# Options*	Kernel Size
Ubuntu	Ubuntu 18.10 Cloud Image	2495 / 5147	62007918
Debloated by Cozart for HTTPD		-51.28%/-99.86%	-65.09%
AWS	Amazon Linux 2	1214 / 946 (929)	58917847
Debloated by Cozart for HTTPD		-29.82%/ -97.46 (-97.42)%	-59.28%
AWS	Firecracker	910 / 0	15136127
Debloated by Cozart for HTTPD		-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

34.91% for Ubuntu and by up to 40.72% for Amazon Linux 2, respectively, for HTTPD as the target application. We will report the detailed results for other applications in Section 7.

We also studied the minimized kernel used by Amazon FireCracker, which is a micro VM specialized for Function as a Service (e.g., AWS Lambda). The Firecracker kernel is based on Linux with handcrafted kernel configuration (910 configuration options), which offers memory overhead of less than 5MB per thread in a guest VM to allow the creation of up to 150 Firecracker instances per second per host. We can see from Table 1 that kernel debloating can still benefit from such a minimized kernel with a reduction of 17.69% using HTTPD as the target application.

4 COZART

Cozart is an advanced OS kernel debloating framework. We use Cozart as a vehicle to understand both the accidental and essential limitations of kernel debloating techniques. Specifically, Cozart integrates our solutions to the accidental limitations in its design, while including support for analyzing the essential limitations.

We chose to develop a new tool instead of building on top of existing tools (e.g., Undertaker [46] and LKTF [39]) because the new design would require substantial architectural changes and does not fit in the current implementation of existing tools.

4.1 Design Principles

Cozart shares the same high-level working principle as the state-of-the-art kernel debloating techniques [25, 46, 47, 52, 69, 71, 79]. It traces the kernel footprint reached by the target application workloads to determine the kernel configuration required by the target application. The debloated kernel will only include those kernel features instead of all available features or features enabled in default configurations. Cozart distinguishes itself from the state-of-the-art kernel debloating techniques [25, 46–48, 52, 69, 71, 79] in the following aspects:

- **End-to-end visibility.** Cozart targets OS kernels used in cloud-based virtual machines. Therefore, Cozart leverages the visibility of the hypervisor to implement end-to-end observations that are able to trace both the kernel boot phase and application workloads. We build Cozart on top of QEMU.
- **Composability.** A key principle behind Cozart is to make kernel configuration *composable*. Cozart divides a kernel configuration into a set of Configlets. A Configlet is a set of configuration options used to boot the kernel on a given deployment environment (e.g., a VM) or a set of configuration options needed by a target application (e.g., HTTPD). The former is named Baselet and the latter is named Applet. Figure 3 shows examples of Baselet of the Firecracker MicroVM and an Applet of HTTPD. Note that a Baselet is not necessary the minimal set configuration to boot on a specific hardware. Instead, it is a set of configuration options that Cozart traces during the boot phase. Therefore, a Baselet is specialized for the hardware that Cozart traces. A Baselet can be composed with one or more Applets that are generated with the same Baselet to generate the final kernel configuration, which can be used to build the debloated kernel.
- **Reusability.** The Configlets can be stored in persistent storage 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: with Cozart, one can conduct kernel tracing and application workload *offline* and maintain the resultant Configlets in databases. The reusability also avoids repetitive tracing and workload runs, making Configlet generation a one-time effort.

```
CONFIG_HYPERVISOR_GUEST
CONFIG_KVM_GUEST
CONFIG_PARAVIRT
CONFIG_VIRTIO
CONFIG_VIRTIO_BLK
CONFIG_VIRTIO_MMIO
CONFIG_VIRTIO_NET
CONFIG_VIRTUALIZATION
...
```

(a) Baselet of Firecracker

```
CONFIG_EPOLL
CONFIG_EXT4_ENCRYPTION
CONFIG_EXT4_FS
CONFIG_EXT4_FS_POSIX_ACL
CONFIG_EXT4_FS_SECURITY
CONFIG_FUTEX
CONFIG_INET
CONFIG_IP_MROUTE
...
```

(b) Applet of HTTPD

Fig. 3. Examples of Configlets.

- **Support for fast application deployment.** The reusability and composability form the foundation of supporting fast deployment in Cozart. Given a deployment environment (e.g., a VM) and the target application(s), Cozart can effectively retrieve the Baselet and Applets and compose them into the kernel configuration. Cozart then builds the debloated new kernel using the generated kernel configuration. Cozart also caches the intermediate build results (e.g., the object files) corresponding to the Configlets and uses the incremental build support provided by the build system.
- **Fine-grained configuration tracing.** Cozart uses Program Counter (PC) based tracing provided by QEMU to identify configuration options based on the corresponding code patterns described in Figure 1. We chose PC tracing based on the observation that function-level tracing (e.g., `ftrace`) cannot deal with configurations inside functions (Figure 1c).

The ability to generate fine-grained Configlets and the composability of Configlets also allows us to compare Applets generated by various test suites and benchmarks and to further customize Baselets based on domain knowledge.

Figure 4 presents the overview architecture of Cozart. Cozart 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 (Figure 4):

- **Offline phase.** Config Tracker is used for tracking the configuration options used by the deployment environment and target application ❶; Configlet Generator is used to generate Configlets (including both Baselets and Applets and store them in Configlet DB ❷).
- **Online phase.** Given the target deployment environment and the target applications, Cozart uses Kernel Composer to generate the kernel configuration by composing the corresponding Baselet and Applets ❸ and then uses Kernel Builder to build the debloated kernel ❹.

4.2 Tracking Configuration Options

Cozart tracks kernel configuration options used during the kernel execution driven by the target application. Cozart is expected to trace every individual application in order to generate the Applet for the application (which will be stored in Configlet DB). Cozart can later compose multiple Applets of different applications during the online phase.

Configuration tracking is done as follows:

(1) Tracing program counters (PCs) of target application(s);
(2) Mapping PCs to source code;
(3) Correlating source code to configuration options.

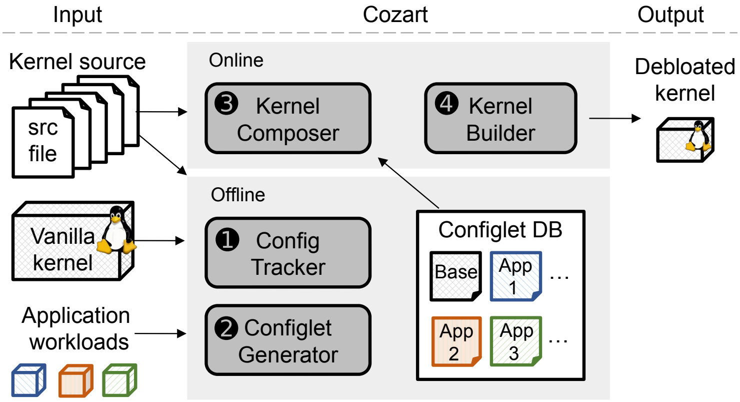

Fig. 4. Cozart overview: Config tracker records configuration options used by the target application. Configlet generator processes these options into Configlets and stores them in Configlet DB. Kernel composer takes Configlets and produces the final configuration. Kernel builder builds the debloated kernel based on the final configuration.

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

Cozart disables Kernel Address Space Layout Randomization (KASLR) to be able to map the addresses to source code correctly. Please note that the disabling of KASLR is only limited to the configuration tracking phase—*the final debloated kernel can still use KASLR*. In other words, this is not a limitation of Cozart.

4.2.2 **Mapping PCs to Source Code Statements.** Cozart uses `addr2line` [51] to map a PC to the corresponding statement in the source code to understand which kernel module is used during the kernel execution. `addr2line` utilizes the debugging information compiled in the kernel binary with the baseline configuration.

Loadable kernel modules (LKMs) need additional handling. An LKM is a standalone binary that can be loaded into a kernel during runtime. Since LKMs are not part of the kernel binary, they are not loaded into memory at a fixed address; therefore, `addr2line` cannot be directly applied. In Cozart, we use `/proc/modules` to obtain the start address of each one of the loaded LKMs and then map the PCs to the statement in the LKM binary.

An alternative is to leverage `localmodconfig`, a kernel utility that outputs current loaded LKMs. However, `localmodconfig` only provides information at the granularity of an entire LKM and thus cannot help debloating an LKM based on fine-grained intra-LKM configuration options.

4.2.3 **Attributing Statements to Configuration Options.** Cozart attributes source code statements to configuration options based on the three patterns presented in Figure 1. We implement text-based analysis based on the semantics of C preprocessors and Makefiles. For the C preprocessor based patterns (Figures 1a and 1b), 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 (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 (cachefiles.o, bind.o and daemon.o) are used.

4.3 Generating Configlets

Baselets and Applets are generated during the phase in which configuration options are tracked. Cozart marks the end of the boot phase using a *landmark program*—a C program that mmaps an empty stub function to a pre-defined "magic address" (we use 0x333333333000 and 0x222222222000 for the start and end) and invokes the stub function. The init script described in Section 4.2.1 calls the landmark program before running the target application. Therefore, Cozart can recognize the (end of) 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 (for instance, 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, e.g., it loads a new device driver on demand (in such cases, the application is hardware-specific).

4.4 Composing Configlets

Cozart composes the Baselet with one or multiple Applets to produce a final kernel configuration that is used to build the kernel. Cozart first unions the configuration options in all the Configlets into an initial configuration and then resolves the dependencies between the configuration options (Section 2.2) using an SAT solver. Cozart does not use KConfig, which would silently deselect unsatisfied dependencies during compilation to ensure a valid configuration. To ensure all the selected configuration options to be included in the final debloated kernel, we model the configuration dependencies as a boolean satisfiability problem and use an SAT solver to generate a final configuration—a valid configuration is one that satisfies all the specified dependencies between configuration options. We follow the SAT-based modeling for kernel configuration options described in [26, 72] and use PicoSAT in the implementation of Cozart [19]. It is possible that the SAT solver returns multiple valid solutions. In this case, our current implementation will use the first solution returned from the solver. Note that the first solution may not strictly be the smallest one among all the returned solutions in terms of size; on the other hand, it is straightforward to compile the kernels for all the returned solutions and select the one for the minimal size.

4.5 Kernel Builder

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

5 STUDY METHODOLOGY

Target Applications. We use six popular open-source server applications, presented in Table 2, that are commonly deployed in virtualized cloud environments. For each application, we use both

Table 2. Applications and their benchmarks and test suites used in our study All the test suites are the official ones shipped with the open-source projects.

Application	Benchmark (Metrics)	Test Suite
HTTPD (v2.4)	Apache benchmark ab (RPS)	Apache-Test [2]
NGINX (v1.15)	Apache benchmark ab (RPS)	NGINX-test [6]
Memcached (v1.5)	Memtier benchmark (RPS)	Memcached/t [5]
MySQL (v5.7)	sysbench (throughput)	MySQL test [11]
PHP (v7.2)	PHP perf. benchmark (execution time)	PHP-test [7]
Redis (v4.0)	Redis benchmark (throughput)	Redis-test [9]

its official test suite and performance benchmarks to drive the kernel, based on which we use Cozart to generate the Configlets.

Performance Benchmarking Setups. We use the official performance benchmarks listed in Table 2 to measure the application performance running in the auto-debloated kernel, in order to verify that the debloating does not introduce any performance regression. All the performance experiments (including the measurement of build times) were performed on a KVM-based VM with 4 VCPUs and 8 GB RAM running on an Intel®Xeon®Silver 4110 CPU operating at 2.10 GHz. We report the benchmark results as the average results of 20 runs.

Baseline Kernel. Our baseline kernel is Linux kernel version 4.18.0 from the latest Ubuntu 18.10 Cloud Image (the most widely used Linux distribution, Section 2). We also repeated all our experiments on Linux kernel version 5.1.9 from Fedora 30 (the top two most popular Linux distributions deployed in cloud environments [10]). We observe very similar results. Therefore, we only present the results from the Ubuntu 18.10 Cloud Image as the baseline.

Besides the Linux kernel, we also applied Cozart to the L4/Fiasco microkernel and Amazon Firecracker's kernel, as discussed in Section 8.

6 DEBLOATING EFFECTIVENESS

We use Cozart to debloat the baseline kernel for the target applications listed in Table 2. Cozart achieves more than 80% kernel reduction (Table 4), which we will discuss in detail in Section 7.2. All the debloated kernel can boot and run application workloads, including the corresponding performance benchmarks and test suites.

To validate that the application workloads running on the debloated kernels have normal runtime behavior, we compare the test results and the log messages generated by the applications running on the debloated kernels and the baseline kernel. We verify that the application behaviors are identical. Additionally, we compared statement-level code coverage results of the applications running on the debloated kernel versus the baseline kernel. We find the coverage results are similar with difference within 1% due to certain non-determinism of test cases. In summary, the debloated kernel does not cause any runtime failure of the target applications.

In the remaining of this section, we report the performance of the debloated kernel generated by Cozart with regards to the baseline kernel in the following aspects: (1) boot time reduction, (2) performance benchmark results, and (3) memory savings.

Boot Time. Boot time is critical to provide elastic runtime environment for the serverless computing. The second column in Table 3 shows the boot time (and the corresponding reduction with regards to the baseline kernel) for each debloated kernel for the target application. We measure

Table 3. Boot time, application performance and memory reduction for debloated kernels. The baseline is the original kernel of the Ubuntu 18.10 Cloud Image. The benchmarks for measuring application performance are described in Table 2. Note that we show both `get` and `set` requests for Redis.

Application	Boot Time (ms)	App. Perf. (%)	Memory 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

the boot time by reading the value from `/proc/uptime` that contains the duration of the system being on since its last restart. The numbers in the table present the average boot time for ten runs. The kernels debloated by Cozart achieve a time reduction of close to 14%. The reduction is attributed to the savings in load time for a smaller sized kernel image and the skipping of module initialization and device registration for the parts that have been removed (e.g., Fuse file system and Hot Plug PCI controller—both of which may have been removed as part of the debloating).

The boot time depends on the Baselet that, in turn, depends on the original kernel as discussed in Section 7.3. A small Baselet will lead to shorter boot times. Surprisingly, Cozart is able to achieve a similar percentage of reduction for Firecracker's kernel with the boot times at less than 100 microseconds.

Application Performance. We benchmark the performance of the applications running on debloated kernels generated by Cozart and compare the benchmark results with the baseline kernel. We do not expect any performance improvements; instead, we look for performance regressions due to removal of performance optimization modules from the original kernels. Note that this is not supposed to happen by the design of Cozart, but we run the performance benchmarks as sanity checks.

Table 3 (the third column) shows the improvements of application performance running on debloated kernels with regards to the baseline. There is no performance regression for any of the applications; instead, the debloated kernels show marginal performance improvements for all the applications, mainly because applications load faster and have smaller memory overheads.

Memory Savings. Smaller memory usages can improve resource utilization. Debloated kernels can have the benefit of memory savings. We define memory savings as the reduction in the memory region that hosts the loaded kernel binary, measured by `MemTotal`, as read from `/proc/meminfo`. The numbers shown in Table 3 are aligned to page sizes (4 KB in our system). We observe about 4 MB memory savings across the debloated kernels for single applications because the kernel size reduction contributes to memory savings, and all applications have similar reduction percentages (which we will discuss in Section 7.1).

To Summarize: The debloated kernels produced by Cozart are all stable—they can directly boot and support the expected application workloads. The debloated kernels can boot 13+% faster than the baseline kernel. The debloated kernel has slightly higher performance compared with the baseline kernel. The debloated kernels achieve more than 4MB memory savings at runtime.

Table 4. Characteristics of the debloated kernel generated by Cozart based on Configlet composition. "Static" and "LKM" refer to all the statically-linked modules and loadable kernel modules, respectively. "Default" refers to a very conservative baseline that only includes the LKMs that are auto-loaded based on Ubuntu 18.10 configuration (Ubuntu includes all the LKMs but does not auto-load every single one by default). "Overall" refers to the overall kernel space (all the modules).

Application	# Remaining Kernel Modules		Kernel Size Reduction			
	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%

7 FINDINGS AND IMPLICATIONS

7.1 Boot Phase Visibility

This section discusses the visibility limitation of the state-of-the-art kernel debloating techniques. It also explains how Cozart addresses this accidental limitation by design.

Kernel debloating has to take care of the kernel boot phase in which the necessary kernel modules are loaded to make sure the kernel can start up and run on the deployment environment; otherwise the kernel could either fail to boot or have performance and security regressions due to missing important options that are loaded during the boot phase.

7.1.1 **Our Experiences with State-of-the-Art Kernel Debloating Tools.** The state-of-the-art tools, as exemplified by Undertaker [46], do not have an automatic solution to tackle the boot phase. This is because existing tools use `ftrace` for kernel tracing, which can only be started after kernel boot. Therefore, existing tools do not have the *visibility* to the kernel boot phase.

To work around this limitation, Undertaker turns on `ftrace` as early as possible by extending the system-provided initial RAM disk (`initrd`) [46]. However, we show that turning on `ftrace` at the initialization of RAM disk is not early enough. As shown in Appendix A, hardware detection happens earlier than mounting RAM disk. We show that disk drivers are not captured in Undertaker, and even the RAM disk support itself has to be whitelisted.

To make the debloated kernel bootable, Undertaker eventually requires users to set a whitelist to ensure certain configuration options are included. However, as articulated in [25], "*that brings the user back to the original configuration issue.*" Given the dynamics of kernel configuration (Figure 2), it is hard for the handcrafted whitelist to keep updated with the kernel evolution.

We use Undertaker to debloat the same Linux kernel presented in the Undertaker paper [46]. Undertaker produces an unbootable kernel. In addition, serial console options (*e.g.*, `CONFIG_SERIAL_8250` and `CONFIG_SERIAL_8250_CONSOLE`) are missing, which makes the kernel print no messages. We manually add back the serial console, and we find that EXT4 filesystem is misconfigured as an LKM (caused by its design bugs). Besides, the SCSI disk support (`CONFIG_BLK_DEV_SD`) is absent

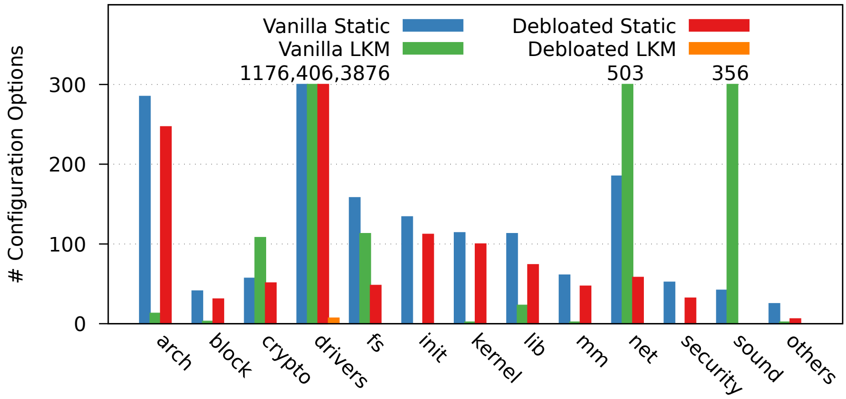

Fig. 5. Boot-phase configuration options in Baselet and in the baseline kernel. We break down the options based on the Linux kernel source tree.

as the current whitelist only preserves an older disk interface, *i.e.*, ATA (`CONFIG_SATA_AHCI`).[3] Our experience is consistent with the other report [25]. It shows that using a handcrafted whitelist is not a sustainable solution.

LKTF [39] attempts to address the invisibility to the boot phase using a search-based approach. LKTF fills in configuration options into the unbootable kernel generated by Undertaker until the kernel can eventually boot. It is reported that LKTF takes about five hours to generate a bootable kernel [25]. Although the debloated kernel generated by LKTF has a higher chance to boot (it is reported that the kernel boots in 1/5 of the time [25]), LKTF involves building and testing a lot of kernels in VMs until one can successfully boot. We are not able to run LKTF because its implementation is hardcoded to a small number of old kernel versions. The kernel generated by LKTF may have reduced performance and security, as LKTF does not guarantee to include all the original security and performance configuration in the debloated kernel.

7.1.2 **Boot-phase Visibility with Cozart.** Cozart addresses the visibility limitation using a lower level tracing provided by the hypervisor, as discussed in Section 4.2. The tracing allows Cozart to observe the complete lifecycle of a kernel and thus include all the modules that are loaded during the boot phase. 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). Figure 5 breaks it down based on the subdirectories in the kernel source tree. The majority of reduction is from `drivers`, `net`, `sound`, and `fs` (file systems), with a reduction of 94+% in total.

Consider drivers as an example. The vanilla kernel aims to support a variety of hardware and thus contains drivers for different devices (*e.g.*, network cards, PCI and media devices). Cozart only includes the one observed in the traces (*i.e.*, those used by the applications), including `acpi`, `scsi`, `cpufreq`, `tty`, `char`, and `dma` drivers. The vanilla kernel contains different filesystems (`fs`) such as FAT and Fuse. Cozart only keeps EXT4 as the filesystem that results in only 48 `fs` modules in the debloated kernel (e.g., `CONFIG_EXT4_FS`, `CONFIG_EXT4_POSIX_ACL`, `CONFIG_EXT4_ENCRYPTION`)

[3] Despite that the kernel debloated by Undertaker does not boot, we still report its overall kernel reduction of 91.38% as a reference.

out of 271 in the baseline. The reduced kernel also includes pseudo-filesystems (*e.g.*, `proc` and `sys`) and other `fs` options (*e.g.*, `CONFIG_FS_IOMAP`, `CONFIG_FS_MBCACHE`) to satisfy dependencies.

> To Summarize: Existing kernel debloating techniques initialize in-kernel tracing methods (*e.g.*, ftrace) too late and cannot observe the boot phase, which is critical to producing a bootable kernel. Cozart leverages the tracing feature provided by the hypervisor to support end-to-end observability (including the complete boot phase and application workloads) and produce stable kernels that can always boot and run applications stably. The debloated kernel by Cozart does not have any performance regression, as shown in Section 6.

7.2 Composability

We show that Configlets can be automatically composed to generate a debloated kernel configuration. A kernel configuration can be composed with one Baselet plus one or more Applets. As described in Section 4.4, Cozart first unions the configuration options in the Configlets and resolves configuration dependencies.

7.2.1 **Composition Experiments.** We evaluate Cozart's composability with the following four types of compositions:

- **Individual application.** We use Cozart to compose each individual Applet and the Baselet to generate a debloated kernel tailored for an individual application.
- **Individual application in a container.** We use Cozart to compose the Applet along with the Configlet of the Docker Runtime (`dockerd`) in conjunction with the Baselets.
- **Multi-application.** We use Cozart to compose a debloated kernel to support the LAMP [4] stack (that includes Apache HTTPD, MySQL and PHP). We use a separate application, a MySQL administration tool with a PHP interface, to validate the LAMP stack works, in addition to each individual application in the stack.

7.2.2 **Applet Characteristics.** Cozart successfully generates the Applets for all of the target applications listed in Table 2. Figure 6 shows the breakdown of the Applets based on the Linux kernel source tree.

All applications have similar distributions across the subdirectories. The average Configlet size is 117, which is ≈1% of the total number of Linux kernel configuration options. The small percentage is expected because (1) most of the options are for hardware features (81+% are hardware options), and (2) applications typically use a small number of system calls and thus have a small kernel footprint. The results also indicate the benefit of decoupling Applets and Baselets—one can rapidly reconfigure the kernel to accommodate new applications as long as we have the corresponding Applets, as shown in Section 7.3.

7.2.3 **Composition Effectiveness.** Table 4 shows the detailed debloating results of the first and second types of composition (single application and application in a container).[4] The reductions are with regard to the baseline kernel. The reason why the reduction numbers look very similar is that 96–99% of the configuration options are from the Baselet generated from the same VM. Cozart can compose the Baselet with the target Applet to achieve 17+% size reduction for static-linked kernel binaries and 86+% size reduction for loadable kernel modules (LKMs).

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

[4] We do not focus on security analysis in this paper and, thus, do not report security metrics such as GenSec and IsoSec [46] or vulnerability reduction [17] (those analysis cannot be automated). We limit ourselves to the understanding that a reduction in total code results in an improved security posture due to reduced attack surfaces.

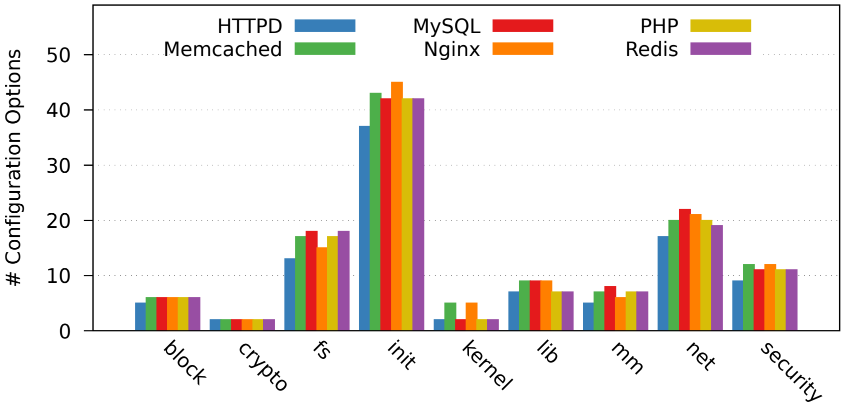

Fig. 6. Characteristics of APPLETS generated by COZART for each target application.

container. We generate debloated kernels by composing multiple APPLETS (including the target application and `dockerd`) and the BASELETS, in a manner similar to how we generate single-application kernels. The debloated kernel can execute all the performance benchmarks described in Table 2. The lower half of Table 4 shows the reduction achieved by debloated Linux kernels running single applications in Docker containers. Specifically, we can see that the `dockerd` CONFIGLET contains most of the APPLETS already. The debloated kernels have similar reduction ratios.

To compose the kernel configuration 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 and run a separate application, `phpMyAdmin` (a MySQL administration tool with a PHP-based interface). We manually exercise the application to ensure the functionalities of all components (*i.e.*, PHP, MySQL and HTTPD) are used. We find no error messages to verify that (1) the kernel supports LAMP stack applications and (2) the kernel supports workloads are different from those used to generate each individual APPLET. We find that `phpMyAdmin` works properly without any dysfunctions. The debloated kernel shows a size reduction of 17.13% for a statically-linked kernel binary and 86.80% for LKMs.

> TO SUMMARIZE: The composability of COZART enables users to generate and maintain application-specific kernel configuration in APPLETS and deployment-specific kernel configuration in BASELETS. A complete kernel configuration can be generated by composing APPLETS and BASELET.

7.3 Fast Application Deployment

A limitation of existing kernel debloating techniques, when one wants to deploy a new application onto a debloated kernel (customized for a different application), is that she has to repeat the entire debloating process from tracing to the re-compilation.[5] Given that debloating a kernel could take hours or days, existing debloating techniques cannot support fast application deployment.

We argue that the aforementioned limitation is caused by the design of existing techniques which make tracing an integral component of the debloating process—the debloating tool has to first trace the kernel by running application workloads and then compile the kernel based on the tracing results. Table 5 compares the tracing time and compilation time. We can see that tracing requires

[5] One can prepare debloated kernels for every combination of applications and thus save repeated tracing effort. However, this requires prohibitive storage costs due to the combinatorial space. For example, for 10 applications, the potential combinations of applications running on a kernel are 2^{10}.

Table 5. The debloating time for tracing and compilation (from scratch and from a previous baseline build).

Application	Tracing Time (second)	Build Time (second)	
		From Scratch	From Baseline
Baseline	0	1035.84	N/A
HTTPD	658.09	119.75	125.47
Memcached	2463.75	119.33	123.92
NGINX	3975.25	119.17	122.52
MySQL	310550.32	119.49	123.37
PHP	14989.02	119.16	123.81
Redis	13004.33	119.57	123.03

Table 6. The number of new options when comparing the new Applet and the composed kernel, and the time it takes to build. Hot/cold refers to hot/cold disk cache.

Applications		# New Modules (Options)	Time (Sec.) (Hot/Cold)
Existing	New		
Apache	PHP	0	0/0
Apache + PHP	Redis	1	5.67/30.73
Apache + PHP	MySQL	2	18.07/64.78
Docker + Apache + PHP	MySQL	0	0/0

significantly more time than compiling. Specifically, the tracing time depends on the application workloads (test suites or benchmarks)—a more complete workloads would lead to higher tracing overhead. For example, MySQL has 5567 test files and requires 86 hours in total to finish the tracing.

With Cozart, tracing can be decoupled from the compilation and thus can be done offline, as long as the results, Configlets, are maintained. *If we have the Configlets of the target applications, debloating a kernel takes less than two minutes if compiled from scratch, as shown in Table 5.*

7.3.1 **Build from Scratch.** As shown in Table 5, building a debloated kernel from scratch takes less than two minutes. The build is more than 8.5× faster than building the baseline kernel, as the debloating skips more than 83% of kernel code (Table 4). The compilation time is proportional to the code size.

We also explore the benefits of *incremental builds* that reusing intermediate files (e.g., object files and LKMs) of a build for the baseline kernel. Our hypothesis is that the build for the debloated kernel should be able to reuse many of the object files and speed up. Surprisingly, we find that such practice does not help speed up the build, but instead slowing down the build slightly, as shown in Table 5. The reason is that the configuration of the debloated kernel is dramatically different from the baseline kernel, and only a few files can be reused—we observe that more than 99% of files need to be re-compiled. On the other hand, when building from a previous build, KBuild needs to check the reusability of each object file, with the overhead being more than the benefit.

7.3.2 **Incremental Builds from Debloated Kernels.** We then explore the benefits of incremental builds from debloated kernels, instead of the baseline kernel. We design a few experiments where we want to add a *new* application on top of an existing debloated kernel and measure the additional build time, as shown in Table 6.

We find that the incremental builds from debloated kernels are very effective. This is mainly attributed to the fact that many applications share common configuration options and only differ in a small number of options. As a result, Cozart only needs to (re-)compile these small number of additional modules, while reusing the majority of the originally built binaries. As shown in Table 6, our framework only needs tens of seconds in the presence of cold caches to rebuild the kernel for supporting a new application, while only a few seconds if the cache is hot (typically when dedicated build servers are in use). The incremental build time is determined by the number and size of the new modules that need to be included. Especially, in two cases, the build is entirely skipped, because the Applet of the new application is included in the current Configlets. Note that this cannot be supported without the exposure and analysis of Configlets—in existing debloating techniques, the same cases have to go through the tracing and building process.

To Summarize: Kernel debloating can be done within tens of seconds if the configuration options of target applications are known. The composability of Cozart allows users to prepare Applets offline. To deploy a new application on a debloated kernel, Cozart composes the Applet of the target application with the Configlets of the current kernel to generate a new debloated kernel within tens of seconds.

7.4 Coverage

One of the essential concerns of applying existing kernel debloating techniques is the completeness of the debloated kernel regarding the kernel footprint of the target applications. If the target application reaches kernel code that is not included in the debloated kernel, the kernel could potentially crash or returns errors to the applications.

The prior studies on kernel debloating techniques implicitly assume that running a benchmark could cover the kernel footprint of the target applications. Taking web servers as the example, prior studies use simple benchmarks that access static documents to generate the debloated kernel [46].

Table 7 shows the statement coverage of each target application when running the official performance benchmark suites. We use `gcov`, an open-source source code coverage analyzer. We can see that the statement coverage reached by the official benchmark is very low—the statement coverage ranges from 6% to 26%. Typically, the benchmarks only exercise the steady states of the target applications, while missing the other part of the application program (e.g., other features, error handling and recovery states). Take web servers such as HTTPD and NGINX as examples. The benchmarks only exercise the code path for serving static web pages and only lead to a statement

Table 7. Statement coverage and number of kernel configuration options of the target applications reached by the official performance benchmarks versus official test suites.

Application	Test Suite		Benchmark	
	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

coverage of 13% and 8% respectively. As a result, the debloated kernel generated based on those benchmarks can hardly support workloads different from the benchmarks.

Test suites can potentially complement benchmarks in exercising the application's kernel footprint, as test suites are designed to exercise different parts of the software and its use cases. Therefore, we explore the feasibility and effectiveness of using test suites for kernel debloating, given that mature software systems usually provide high-coverage test suites, as shown in Table 2.

As shown in Table 7, test suites provide much higher code coverage compared with the benchmarks. However, we can see that the code coverage of the test suites is not perfect. For the open-source applications evaluated in this paper, the statement coverage varies from 29%–73%. It is reported, however, that test suites for industrial software have much higher code coverage. For example, Google reports that the code coverage of their software projects are above 80% [34, 65]. We conclude that lack of high-coverage tests is an essential limitation for the adoption of existing kernel debloating techniques based on dynamic tracing. We believe that automated testing techniques such as fuzzing testing [44, 53] and concolic testing [30, 66] and other symbolic execution based test generation techniques [22, 23] can effectively improve the test coverage and alleviate the limitations. On the other hand, those test auto-generation techniques are not silver bullets, *e.g.*, fuzz testing provides few guarantees on coverage, while symbolic execution is challenging to scale.

We compare the Applets generated from benchmarks versus test suites. Table 7 shows the size of the two Applets for each application. In general, test suites have higher code coverage and lead to the larger size of Applets, compared with benchmarks. The only exception is HTTPD. The Applet generated from the benchmark has 23 more options, despite lower code coverage. We attribute this to HTTPD's poor test suite.

Interestingly, we find that the configuration options discovered by benchmarks are not always included in the Applets from test suites even for high-coverage test suites such as Memcached. In Memcached, there are three options that only exist in Applets generated by the benchmark. One of them is CONFIG_PROC_SYSCTL, which is captured when Memcached reads the maximum number of file descriptors to check whether it can allocate more connections in the scenario where the number of requests overloads the Memcached server.

We would like to note that code coverage cannot guarantee the coverage of the target application's kernel reach (the kernel features needed to support the application). A test suite with 100% statement coverage can cover all possible system calls invoked by the target application, but provides no guarantee to cover all possible system call argument values, or all possible kernel states. Hence, the limitations of dynamic tracing is essential (not accidental) to existing kernel debloating techniques.

To Summarize: An essential limitation of using dynamic-tracing based techniques for kernel debloating (which are the *de facto* approach) is the imperfect 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 the limitation to a certain extent. On the other hand, code coverage cannot guarantee the coverage of the target application's kernel reach.

7.5 Using Domain-specific Information

One essential limitation of existing kernel debloating techniques is the inability to distinguish between kernel modules which are *executed* during the tracing versus those that are *needed*. Current techniques (including Cozart) keep all the kernel modules/code that are used during the boot phase or during application workloads in the debloated kernel. However, modules/code that are executed are not necessarily needed—it is possible that they are reached only because they happen

Table 8. Boot-related kernel reduction on top of the Baselet using domain-specific information (the `kvmconfig` and `xenconfig` templates provided by the Linux kernel).

	Static Kernel Size	# Remaining Kernel Modules	
		Static	LKM
Original debloated kernel	20579167	1215	7
KVM	12302255 (-40.22%)	680 (-44.03%)	1 (-85.71%)
XEN Guest (DomU)	12524616 (-39.14%)	709 (-41.65%)	2 (-71.43%)

to exist on the execution paths. For example, for virtual machines (VMs), kernel modules/code for BIOS, thermal control, power management are executed but are not needed to boot the VM.

In this section, we study the opportunities of further debloat the kernel based on domain knowledge to remove the kernel modules/code that are not needed for the deployment. We study KVM-based VMs (such as Amazon Firecracker) and Xen VMs. As described in Section 2.3, the Linux source code provides configuration templates, `kvmconfig` and `xenconfig`, which provide domain-specific information for KVM and Xen respectively. For example, `kvmconfig` uses `virtio` as the I/O interface for both net and block devices, while `xenconfig` uses the `Xen` front-end (`Dom0`) as the main I/O interface; the original Baselet also uses other I/O interfaces such as the SCSI disk driver which are not needed for KVM and Xen VMs.

We use these two templates to replace the original Baselets to further debloat the already-debloated kernel for KVM (we use Amazon Firecracker as the KVM frontend) and Xen, respectively. Table 8 shows the results. *We observe a 40+% and 39+% kernel size reduction of KVM and Xen compared with the original Baselet, which is a result of 40+% and 41+% reduction in the number of configuration options for KVM and Xen, respectively.*

The significant reduction comes from the fact that many kernel modules (including partition types, processor features, BIOS, thermal control, power management, ACPI, input devices and legacy serial support) are executed by the baseline kernel at the boot time and thus are included by Cozart. However, those modules are not needed if the deployment is KVM or Xen virtual machines. Therefore, with domain-specific information of these VMs, we can further remove those modules.

To Summarize: Domain-specific information can be used to further debloat the kernel by removing the kernel modules that are executed in the baseline kernel but are not needed by the 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 provided by the Linux kernel source.

8 GENERALITY

To the best of our knowledge, all existing kernel debloating tools and studies only focus on the generic Linux kernel. We report our experience in debloating other types of OS kernels by porting Cozart to the L4/Fiasco microkernel and the Firecracker kernel.

8.1 Cozart Portability

Cozart is portable to other types of OS kernels. We have ported Cozart to debloat the L4/Fiasco microkernel and the Firecracker kernel. We apply Cozart on the L4/Fiasco microkernel that also uses KConfig as the configuration language with minor changes to a newer source tree layout and different build steps. We modify the kernel bus from MMIO to PCI for QEMU tracing and change the bus back to boot on Firecracker.

Our experience shows that configuration-driven kernel debloating techniques are independent of kernel architectures, regardless of whether they are monolithic kernels or microkernels. The porting only requires the integration with the kernel configuration system (e.g., KConfig) and the kernel build system (e.g., KBuild).

8.2 L4 Microkernel

We apply Cozart to the L4/Fiasco microkernel [3]. The result is surprising—the debloated kernel is 47% smaller when compared to the default. The reason for the significant reduction is that the default L4/Fiasco kernel configuration enables a heavy kernel debugger, `CONFIG_JDB`, that contributes to a big part of the final kernel. The debloated kernel no longer houses this module. We would like to note that it may sound straightforward that removing a debugger makes the kernel smaller in size; however, without an automated solution like Cozart, it is impractical to examine and select every single configuration option manually, given the hundreds and ever-increasing number of configuration options (L4/Fiasco has 218 configuration options). This has been repeatedly validated by recent user studies on system configuration management [33, 76–78].

We choose to run an ambient occlusion renderer benchmark, aobench [18], as none of the applications in Table 2 can directly run on L4/Fiasco. We run aobench in a KVM-based VM with 1 VCPU and 128 MB RAM and observe a performance improvement of 4%.

8.3 Firecracker

While still based on Linux, Amazon Firecracker's kernel is a special case, as the kernel is extensively minimized to optimize for Function-as-a-Service workloads (Firecracker does not support generic Linux kernel). We apply Cozart on the Firecracker kernel using the target applications listed in Table 2 to understand the space for application-specific kernel debloating like Cozart to benefit an extensively optimized kernel like 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 microseconds to 92.5 microseconds).[6]

To Summarize: Application-oriented kernel debloating can lead to further kernel code reduction for microkernels (e.g., L4) and extensively customized kernels (e.g., the Firecracker kernel). The reduction is significant: 47.0% for the L4 microkernel and 19.76% for the Firecracker kernel.

9 RELATED WORK

Much research has shown the effectiveness of configuration-driven kernel debloating regarding the reduction of attack surfaces [46, 71], security vulnerabilities [17] and many others [13–15]. Kurmus et al. [46] provide a formal definition of *the attack surface* and devise two models for quantifying the attack surface as a security metric. Their measurement based on two server use cases shows that configuration-driven kernel debloating can reduce the attack surface by 50%–85% under both models. Alharthi et al. [17] conduct a comprehensive analysis of 1530 Linux kernel vulnerabilities documented in the National Vulnerability Database since 2005. The analysis reports that 89% of the vulnerabilities can be nullified by configuration, and 34%–74% of them can be avoided if configuration-driven debloating is applied. Our work is complementary to the aforementioned

[6] We use Firecracker's own mechanism to measure the boot time instead of reading from `/proc/uptime`. Firecracker uses a pre-defined I/O port to notify boot completions.

studies. The focus of this paper is to study the practicality of configuration-driven kernel debloating, with the goal of realizing the security and performance benefits in real-world deployments.

Our work mainly focuses on configuration-driven debloating techniques, which are the *de facto* method for debloating OS kernels [25, 38, 39, 46–48, 52, 69, 71, 79]. Specifically, most open-source kernel debloating tools, the artifacts we studied, use configuration-driven techniques. There have been a number of software debloating techniques based on program analysis, compiler optimizations, and machine learning. However, those approaches only target user-space applications [24, 32, 36, 61–63, 67, 70] and currently cannot work with OS kernels due to the limitations of analysis techniques (*e.g.*, the scalability and soundness of binary and source code analysis). A series of ongoing efforts have been undertaken in the Linux kernel community to carve the kernel for small environments, including garbage collection and link-time optimization [59, 60], as well as manually debloating based on the hardware [57, 58]. However, the motivation to shrink the kernel and the security implication (e.g., attack surface reduction) is unclear.

Unikernels (also known as Library OS) [28, 41, 43, 49, 50, 55, 73] eliminate the privilege barrier between kernel and user spaces and have a standalone image that contains both the kernel and application. Unikernels have a very small kernel because the kernel code is only included when the application includes it explicitly. However, unikernels are not universal because of the overheads of porting applications that rely on Linux system call APIs—it requires non-trivial developer time and expertise. Cozart is mainly designed for monolithic kernels such as Linux and FreeBSD which are the *de facto* kernels used in practice, despite recent advances in microkernel and unikernel research (discussed in Section 3). We have not experimented Cozart with unikernels. In principle, we believe that Cozart can still benefit unikernels as long as the unikernel exposes a configuration system to support the configurability of the target applications in a similar vein as how Cozart works with microkernels (Section 8.2).

10 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 requirements for manual intervention. This paper studies the practicality of OS kernel debloating techniques with the goal of making kernel debloating practical in real-world deployments. We identify the limitations of existing kernel debloating techniques that hinder their practical adoption. To understand these limitations, we build Cozart, an automated kernel debloating framework which enables us to conduct a number of experiments on different types of OS kernels with a wide variety of applications. We share our experience and solutions in addressing the accidental limitations that exist in the existing kernel debloating techniques and discuss the challenges and opportunities in addressing the essential limitations that are critical to the practicality of OS kernel debloating.

We have made Cozart and the experiment data publicly available at:

https://github.com/hckuo/Cozart

ACKNOWLEDGEMENT

We thank the anonymous reviewers of SIGMETRICS and our shepherd, Bhuvan Urgaonkar, for the valuable feedback that improved this paper. We thank Neil Zhao for the discussion of test generation and coverage which improved Section 7.4. We thank the System Reading Group at UIUC for useful feedback that improved the presentation of an early draft.

REFERENCES

[1] Amazon Linux 2. https://aws.amazon.com/amazon-linux-2.

[2] Apache-Test. http://perl.apache.org/Apache-Test/.

[3] FIASCO : The L4Re Microkernel. http://os.inf.tu-dresden.de/fiasco.

[4] LAMP. http://ampps.com/lamp.

[5] Memcached Test. https://github.com/memcached/memcached/tree/master/t.

[6] nginx-tests. https://github.com/nginx/nginx-tests.

[7] PHP Test. https://github.com/php/php-src/tree/master/tests.

[8] QEMU - the FAST! processor emulator. https://www.qemu.org.

[9] Redis Test. https://github.com/antirez/redis/tree/unstable/tests.

[10] the cloud market. https://thecloudmarket.com/stats#/by_platform_definition.

[11] The MySQL Test Suite. https://dev.mysql.com/doc/refman/5.7/en/mysql-test-suite.html.

[12] Configuring the FreeBSD Kernel. https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig-config.html, 2019.

[13] Iago Abal, Claus Brabrand, and Andrzej Wasowski. Variability Bugs in the Linux Kernel: a Qualitative Analysis. In *ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, Vasteras, Sweden, 2014.

[14] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. In *ACM Transactions on Software Engineering and Methodology (TOSEM'18)*, 2018.

[15] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Oliver Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Technical Report hal-02314830, INRIA, October 2019.

[16] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. Learning From Thousands of Build Failures of Linux Kernel Configurations. Technical report, INRIA, June 2019.

[17] Mansour Alharthi, Hong Hu, Hyungon Moon, and Taesoo Kim. 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.

[18] aobench. Ambient Occlusion Benchmark. https://github.com/gnzlbg/aobench, 2019.

[19] Armin Biere. Picosat essentials. *JSAT*, 4, 2008.

[20] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "Micro" Back in Microservice. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, MA, USA, July 2018.

[21] Brendan Burns and David Oppenheimer. Design Patterns for Container-based Distributed Systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, CO, USA, June 2016.

[22] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, December 2008.

[23] Cristian Cadar and Koushik Sen. Symbolic Execution For Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, February 2013.

[24] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. TOSS: Tailoring Online Server Systems through Binary Feature Customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'18)*, Toronto, Canada, October 2018.

[25] Jonathan Corbet. A different approach to kernel configuration. https://lwn.net/Articles/733405/, September 2016.

[26] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 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, 2012.

[27] Alexia Emmanoulopoulou. infographic: How many people use Ubuntu? https://blog.ubuntu.com/2016/04/07/ubuntu-is-everywhere, April 2016.

[28] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, USA, 1995.

[29] Kai Germaschewski and Sam Ravnborg. Kernel configuration and building in Linux 2.5. In *Proceedings of the 2003 Linux Symposium*, Ottawa, Ontario, Canada, July 2003.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, June 2005.

[31] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'19)*, Asilomar, California, USA, January 2019.

[32] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, Toronto, Canada, 2018.

[33] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 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.

[34] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code Coverage at Google. In *Proceedings of the 2019 12th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2019)*, Tallinn, Estonia, 2019.

[35] MSV Janakiram. 10 Reasons Why Ubuntu Is Killing It In The Cloud. https://www.forbes.com/sites/janakirammsv/2016/01/12/10-reasons-why-ubuntu-is-killing-it-in-the-cloud, January 2016.

[36] Yufei Jiang, Dinghao Wu, and Peng Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, 2016.

[37] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[38] Junghwan Kang. A Practical Approach of Tailoring Linux Kernel. In *The Linux Foundation Open Source Summit North America*, Los Angeles, CA, September 2017.

[39] Junghwan Kang. An Empirical Study of an Advanced Kernel Tailoring Framework. In *The Linux Foundation Open Source Summit*, Vancouver, BC, Canada, August 2018.

[40] Junghwan Kang. Linux Kernel Tailoring Framework. https://github.com/ultract/linux-kernel-tailoring-framework, August 2018.

[41] Antti Kantee and Justin Cormack. Rump Kernels: No OS? No Problem! *;login:*, 39(5):11–17, 2014.

[42] kernel.org. Kconfig. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt, 2018.

[43] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSV – Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, Philadelphia, PA, USA, June 2014.

[44] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, Toronto, Canada, 2018.

[45] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B. Bobba, David Lie, and Jesse Walker. MultiK: A Framework for Orchestrating Multiple Specialized Kernels. *arXiv:1903.06889*, March 2019.

[46] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 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.

[47] Che-Tai Lee, Zeng-Wei Hong, and Jim-Min Lin. Linux Kernel Customization for Embedded Systems By Using Call Graph Approach. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, Kitakyushu, Japan, January 2003.

[48] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An Application-Oriented Linux Kernel Customization for Embedded Systems. *Journal of Information Science and Engineering*, 20(6), 2004.

[49] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, Houston, Texas, USA, 2013.

[50] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the Virtual Library Operating System. *Communications of the ACM*, 57(1):61–69, 2014.

[51] Linux man page. addr2line(1). https://linux.die.net/man/1/addr2line, 2019.

[52] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 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.

[53] Valentin J.M. Manés, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, , Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *arXiv:1812.00140*, April 2019.

[54] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, Hyderabad, India, 2014.

[55] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*,

Providence, Rhode Island, USA, April 2019.

[56] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. A Study of Feature Scattering in the Linux Kernel. In *IEEE Transactions on Software Engineering (TSE)*, 2018.

[57] Nicolas Pitre. LWN: Shrinking the kernel with a hammer. https://lwn.net/Articles/748198/, 2018.

[58] Nicolas Pitre. LWN: Shrinking the kernel with an axe. https://lwn.net/Articles/746780/, 2018.

[59] Nicolas Pitre. LWN: Shrinking the kernel with link-time garbage collection. https://lwn.net/Articles/741494/, 2018.

[60] Nicolas Pitre. LWN: Shrinking the kernel with link-time optimization. https://lwn.net/Articles/744507/, 2018.

[61] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, August 2019.

[62] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA, August 2018.

[63] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, Paderborn, Germany, 2017.

[64] Neil Savage. Going Serverless. *Communications of the ACM*, 61(2), February 2018.

[65] Alberto Savoia. Code coverage goal: 80% and no less! https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html, July 2010.

[66] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, Lisbon, Portugal, September 2005.

[67] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, Montpellier, France, September 2018.

[68] Steven She and Thorsten Berger. Formal Semantics of the Kconfig Language. Technical report, Electrical and Computer Engineering, University of Waterloo, Canada, January 2010. Technical Note.

[69] Klaus Stengel, Florian Schmaus, and Rüdiger Kapitza. EsseOS: Haskell-based Tailored Services for the Cloud. In *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware (ARM'13)*, Beijing, China, December 2013.

[70] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided Program Reduction. In *In Proceedings of the 40th International Conference on Software Engineering (ASE'18)*, 2018.

[71] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Automatic OS Kernel TCB Reduction by Leveraging Compile-time Configurability. In *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep'12)*, Hollywood, CA, 2012.

[72] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the Sixth Conference on Computer Systems (Eurosys'11)*, Salzburg, Austria, April 2011.

[73] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, Amsterdam, The Netherlands, 2014.

[74] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 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, 2016.

[75] Bart Veer and John Dallaway. The eCos component writer's guide. *Available: ecos. sourceware. org/ecos/docs-latest/cdl-guide/cdlguide. html*, 2000.

[76] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 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.

[77] Tianyin Xu, Vineet Pandey, and Scott Klemmer. An HCI View of Configuration Problems. *arXiv:1601.01747*, January 2016.

[78] Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR)*, 47(4), July 2015.

[79] Lamia M. Youseff, Richard Wolski, and Chandra Krintz. Linux Kernel Specialization for Scientific Application Performance. Technical Report 2005-29, University of California Santa Barbara, 2005.

A KERNEL INITIALIZATION

The following code snippet is a simplified version of the Linux kernel initialization procedure. It presents evidence that to start tracing from RAM disk (as used by Undertaker [46]) is too late. RAM disks are only executed (Line 26) after tens of subsystems finish initialized. The detailed discussion can be found in Section 7.1.1.

```
 1  /* linux5.1.9/init/main.c */
 2  static int __ref kernel_init(void *unused)
 3  {
 4      int ret;
 5      ...
 6
 7      workqueue_init();
 8      init_mm_internals();
 9      do_pre_smp_initcalls();
10      lockup_detector_init();
11      smp_init();
12      sched_init_smp();
13      page_alloc_init_late();
14      cpuset_init_smp();
15      shmem_init();
16      driver_init();
17      init_irq_proc();
18      do_ctors();
19      usermodehelper_enable();
20      do_initcalls();
21
22      ...
23      integrity_load_keys();
24      load_default_modules();
25      ...
26      run_init_process(ramdisk_execute_command); /* Too late! */
27      ...
28  }
```