HACKSAW: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis

Zhenghao Hu*
New York University

Sangho Lee
Microsoft Research

Marcus Peinado
Microsoft Research

ABSTRACT

Kernel debloating is a practical mechanism to mitigate the security problems of the operating system kernel by reducing its attack surface. Existing kernel debloating mechanisms focus on specializing a kernel to run a target application based on its dynamic traces collected in the past—they remove functions from the kernel which are not used by the application according to the traces. However, since the dynamic traces do not ensure full coverage, false removals of required functions are unavoidable. This paper proposes HACKSAW, a novel mechanism to debloat a kernel for a target machine based on its hardware device inventory. HACKSAW accurately debloats a kernel without false removals because figuring out which hardware components are attached to the machine as well as which device drivers manage them is comprehensive and deterministic. HACKSAW removes not only inoperative device drivers that do not control any attached hardware components but also other kernel modules and functions which are associated with the inoperative drivers according to three dependency analysis approaches: call-graph, driver-model, and compilation-unit analyses. Our evaluation shows that HACKSAW effectively removes inoperative kernel modules and functions (i.e., their respective reduction ratios are 45% and 30% on average) while ensuring validity and compatibility.

ACM Reference Format:
Zhenghao Hu, Sangho Lee, and Marcus Peinado. 2023. HACKSAW: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In *Proceedings of Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Securing the operating system kernel against potential attacks is challenging. Existing monolithic kernels for general-purpose computing machines (e.g., Linux and Windows kernels) are extremely complex in order to support diverse user software through system calls and various hardware devices through kernel device drivers. The Linux kernel currently provides around 300 system calls and supports around 9,000 device drivers [41]—making it free from security vulnerabilities is almost impossible. For example, more than 1,000 and 2,500 CVEs have been reported in the Linux kernel and Windows 10 from 2018 to 2022, respectively [23, 24]. To this end, kernel developers and researchers have proposed various mechanisms to reduce the kernel's attack surface (e.g., system call filtering [85] and resource isolation [72]) as well as to detect and eliminate its bugs (e.g., fuzzing and sanitization [82]).

Software-based kernel debloating [2, 36, 37, 40, 45–47, 53, 87, 88] is a popular approach to reduce the kernel attack surface. It is based on an empirical observation that a general-purpose kernel provides many functions or features (e.g., system calls, network protocols) that a certain application does not use at all. Thus, if it only needs to run a target application, it specializes a kernel for the application by removing the kernel functions that the application does not use. In general, it deals with unnecessary kernel functions in three steps. First, it runs a target application in a certain environment with given test cases to collect the application's execution traces related to kernel functions. Second, it analyzes the collected execution traces to identify unused kernel functions. Third, it removes them from the kernel to secure run the target application in the future.

Unfortunately, the software-based kernel debloating has a fundamental limitation: its accuracy heavily depends on the coverage of test cases. Narrow test cases fail to make a target application interact with enough kernel functions (e.g., deep, stateful, or environment-specific functions), resulting in incomplete execution traces and thus false removals of necessary functions [45, 54]. Executing the target application with such an over-debloated kernel can result in application or system crashes. Several mechanisms [36, 88] mitigate this problem to some extent (e.g., maintaining fallback paths), but false removal is still unavoidable.

In this paper, we consider the kernel debloating problem from a completely different angle—we specialize a kernel for a target machine which runs the kernel based on its hardware components. This new angle allows us to accurately debloat the kernel without false removals. Modern machines provide comprehensive and deterministic mechanisms to identify which hardware components are attached to them [86]. This results in a list of hardware identifiers (IDs) and descriptions which we call *device inventory*. We can use the device inventory to figure out whether certain device drivers are necessary. This is accurate because, unlike the system calls for arbitrary applications, device drivers are written for specific hardware devices they support which are well specified using hardware IDs. Further, we deal with not only (unnecessary) device drivers but also other kernel modules and functions that *depend on* the drivers in order to be loaded or operate. For example, a recent attack against the `rdma-core` package [35] exploits the vulnerabilities residing in iSCSI kernel modules which can be triggered when they interact with an InfiniBand device driver even when there is no InfiniBand hardware. Also, a recent attack against the `vsock`

*Work done while this author was an intern at Microsoft Research.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

kernel module [66] can be triggered even if there is no underlying hypervisor providing a corresponding virtual socket device.

We propose Hacksaw, a hardware-centric kernel debloating mechanism based on device inventory and dependency analysis. Hacksaw identifies and removes *inoperative* kernel functions based on a target machine's device inventory including CPU, PCI, USB, and other peripheral devices that are attached to the machine (both physical and virtual devices). Without the corresponding hardware components, these kernel functions do not properly operate. Through dependency analysis, Hacksaw discovers and removes those inoperative kernel functions including not only device drivers which directly manage certain hardware components but also other kernel modules and functions which are associated with the device drivers. This allows Hacksaw to mitigate both hardware-driven and software-driven vulnerabilities which can be triggered even without actual hardware components.

Hacksaw consists of four procedures: 1) identify device drivers within a target operating system kernel, 2) construct a hardware-centric dependency graph based on the identified drivers and the offline analysis of the target kernel, 3) probe a target machine's hardware components, and 4) remove inoperative kernel functions from the target kernel using the graph and the probing result.

Hacksaw first identifies the separate and built-in drivers that a target kernel supports based on whether they contain hardware IDs for device matching (i.e., vendor and device IDs) [34, 65].

Next, Hacksaw constructs a hardware-centric dependency graph which is a directed graph among kernel functions related to the identified device drivers. Hacksaw initializes the graph using the identified drivers' functions as vertices while creating directed edges based on their dependencies. Then, it analyzes other kernel functions not in the graph to check whether they depend on any function in the graph and whether any function in the graph depends on them. If they do, it adds them into the graph while creating directed edges accordingly. Hacksaw repeats this until there are no more kernel functions to add. Hacksaw considers three kinds of dependency information: 1) call-graph information (i.e., function caller or callee), 2) driver-model information (i.e., driver bus and device class dependency), and 3) compilation-unit information (i.e., build system and configuration dependency). While constructing the graph, Hacksaw ignores any indirectly callable functions (e.g., functions whose addresses are taken) to avoid false removals.

Then, Hacksaw compiles the device inventory of a target machine by probing attached hardware components through the kernel and firmware (BIOS/UEFI) and, finally, it uses both the hardware-centric dependency graph and the device inventory to identify and remove inoperative kernel functions. In particular, it first marks functions in the graph which are associated with the inoperative device drivers according to the device inventory. Then, it propagates the inoperative marks throughout the graph based on the dependency and propagation rules: 1) mark a function if it depends on the marked functions and 2) mark a function if all functions depending on it are marked. In the end, it removes all marked functions by deleting corresponding driver or module files and rewriting corresponding functions in the kernel image or modules. It does not delete any functions not in the graph to avoid false removals.

We prototype Hacksaw for the Linux kernel. We use LLVM [48] for the static analysis of Linux kernel source code and kmax [28] to analyze its build system—Kbuild [11] and Makefile [10]. We further use Z3 [19] to identify build configuration dependencies between object or module files. In addition, we write Python and shell scripts for various other tasks including hardware probing, graph analysis, binary patching, and file removal.

We evaluate Hacksaw on popular environments with predictable hardware configurations: virtual machine instances in three public cloud services (Amazon, Azure, and Google) and two hypervisors (Hyper-V and KVM). We run Hacksaw against 12 different Linux system images (five bare-metal images and seven cloud images) to debloat them according to the probed hardware configurations. Note that we use these virtual machine instances because they have deterministic hardware profiles and are widely used. Hacksaw works for any bare-metal and virtual machines. We confirm that, on average, Hacksaw removes 45.0% of the kernel module files in the system image (i.e., root file system) and 30.0% of the functions in the kernel images. This removal mitigates 47.4% of CVEs related to drivers or modules. Further, we empirically confirm that Hacksaw does not introduce compatibility problems. It runs seven real-world test applications from the Phoronix Test Suite [64] without errors and passes all Linux Test Project (LTP) tests [50].

This paper makes the following contributions:

- Hacksaw is the first system which debloats the kernel based on the device inventory and dependency with no false removal.
- Hacksaw comprehensively and systematically analyzes the call-graph, driver-model, and compilation-unit dependency of the Linux kernel based on device drivers.
- Hacksaw effective reduces kernel attack surface and mitigates vulnerabilities while ensuring validity and compatibility.

Hacksaw is available at https://github.com/microsoft/Hacksaw.

2 BACKGROUND: DEVICE DRIVER MODEL

This section explains the Linux device driver model [17, 83]. Other operating systems, such as Windows, have a similar driver model.

2.1 Basic Entities

Hardware device. A hardware device is a physical or virtual peripheral device attached to a computing machine. It embeds a unique hardware ID [34, 65] to be identified and managed by the kernel or firmware. For example, the configuration space of PCI devices contains the vendor ID and device ID registers [62]. The kernel uses a hardware discovery mechanism, such as the Advanced Configuration and Power Interface (ACPI) [86] and the Open Firmware (OF) device tree [20], to retrieve such information.

Hardware bus. A hardware bus is a hardware channel to connect peripheral devices to a machine. A machine typically has multiple buses with different types (e.g., PCI and USB) to manage devices with specific hardware interfaces. Once a machine boots up, the kernel enumerates every bus—with the help of BIOS/UEFI and ACPI—to detect every device attached to the machine.

Device driver. A device driver is a piece of kernel code to interact with devices to control them or exchange data with them. It has an ID table containing the hardware IDs of devices that it supports for device matching. A device driver can have different ID tables if it supports devices belonging to different buses.

Bus driver and type. A bus driver is a special device driver to manage a certain hardware bus as well as devices and drivers belonging to the bus. If the kernel recognizes a hardware bus, it loads the corresponding bus driver to register the bus with the kernel (i.e., invoke `bus_register()`) to create the kernel data structures for each bus type including lists of detected devices and loaded drivers.

Device class. A device class is a logical group of hardware devices providing similar functionalities and interfaces like input, InfiniBand, and Trusted Platform Module (TPM) regardless of their hardware buses. This abstraction allows the kernel to control similar devices with unified kernel functions. Typically, a class driver (or a library driver for class management) registers a new class (i.e., invoke `class_register()`) and defines a function for assigning a device to a certain device class, which will be invoked by drivers during their initialization or hardware probing.

2.2 Device/Driver Binding and Registration

Driver binding. Driver binding is a procedure to associate a device with a device driver [42]. If the kernel detects a device during bus enumeration, it looks up a device driver that can handle the device. To this end, it retrieves the hardware ID from the device and then checks device drivers (for the corresponding bus type) in the system to see whether the retrieved hardware ID matches an entry of their ID tables. It binds the device and the driver if their hardware IDs are matched. This device binding can be done with the device drivers already loaded in the kernel memory or device drivers in the file system (i.e., Loadable Kernel Modules (LKMs)) which can be automatically loaded according to their `modalias` [75].

Driver entry point. A device driver has an entry-point or initialization function to register itself with the kernel. The entry-point function of a device driver is invoked by the kernel when it loads the driver at the end of driver binding. To enable this invocation, the entry-point function is exported to the outside with a specific or predictable symbol (e.g., `init_module()` for LKM drivers) such that the kernel can easily find and call it. In addition, some device drivers perform hardware checks and initialization inside their entry point functions.

Device/driver registration. Devices and drivers which are bound and initialized are registered with the kernel core. Two main kernel functions for this registration are `device_register()` (or `device_add()`) and `driver_register()`. `device_register()` expects a device structure containing device information, bus type, and device class as an argument. `driver_register()` expects a driver structure containing driver information and bus type as an argument.

Instead of invoking these low-level registration functions directly, device drivers rely on custom registration functions defined in the corresponding bus and class drivers which further define the bus type and device class data structures, respectively. For example, during initialization, USB drivers invoke `usb_register_driver()` defined in the USB bus driver, `usbcore`, which eventually calls `driver_register()` to register the drivers with the kernel along with `usb_bus_type` defined in `usbcore` (Figure 1). InfiniBand drivers invoke `ib_register_device()` (e.g., in the probe function) defined in the InfiniBand class driver, `ib_core`, which eventually calls `device_add()` to register the device with the kernel with `ib_class` defined in `ib_core` (Figure 2).

```c
/* drivers/usb/core/driver.c */
struct bus_type usb_bus_type = { /* USB bus type */
  .name   = "usb",
  .match  = usb_device_match,
  .uevent = usb_uevent,
  .need_parent_lock = true,
};
int usb_register_driver(struct usb_driver *new_driver,
  struct module *owner, const char *mod_name) {
  ...
  new_driver->drvwrap.driver.bus = &usb_bus_type; /* specify bus type */
  retval = driver_register(&new_driver->drvwrap.driver);
  ...
}
EXPORT_SYMBOL_GPL(usb_register_driver);

/* drivers/usb/core/usb.c */
static int __init usb_init(void) { /* USB bus entry point */
  ...
  retval = bus_register(&usb_bus_type);
  ...
}
subsys_initcall(usb_init);
```

(a) Part of USB bus driver (usbcore) code.

```c
/* drivers/bluetooth/ath3k.c */
static const struct usb_device_id ath3k_table[] = { /* ID table */
  { USB_DEVICE(0x0CF3, 0x3000) },
  ...
};
static struct usb_driver ath3k_driver = { /* USB driver structure */
  .name       = "ath3k",
  .probe      = ath3k_probe,
  .disconnect = ath3k_disconnect,
  .id_table   = ath3k_table,
  .disable_hub_initiated_lpm = 1,
};
module_usb_driver(ath3k_driver); /* entry point */

/* include/linux/usb.h */
#define usb_register(driver) \
  usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME)
#define module_usb_driver(__usb_driver) \
  module_driver(__usb_driver, usb_register, usb_deregister)
```

(b) Part of USB Bluetooth device driver (`ath3k`) code.

Figure 1: Driver registration with the USB bus.

2.3 Driver Compilation and Loading

Separate and built-in driver. A device driver can be compiled as a separate file (i.e., an LKM) or statically compiled into the kernel image (i.e., a built-in driver). If a device driver is compiled as an LKM, its ID tables are translated into `modalias` [71], like `pci:v00001002d00001304sv*sd*bc*sc*i*` for an AMD GPU (vendor ID: `0x1002`, device ID: `0x1304`), and stored in the module's metadata area. An LKM driver can be dynamically loaded into the kernel memory due to driver binding, system configuration (e.g., `systemd-modules-load.service`), or user request (e.g., `modprobe`).

A built-in driver is embedded into the kernel image, so it is loaded and initialized during the early boot phase. Since the built-in driver does not separately maintain metadata (i.e., it does not have `modalias`), figuring out its hardware IDs is challenging. Also, to avoid symbol conflicts (i.e., having multiple `init_module()` in the kernel image), the name of the entry-point function of every built-in driver is mangled. For example, the Linux kernel mangles built-in entry-point function names based on the original driver names and the entry-point function names (§5.1), like in this XenBus entry-point function `__initcall__kmod_xenbus__291_1067_xenbus_init2()`.

Platform device and driver. A platform device is a special hardware device whose driver must be statically compiled into the kernel

```c
/* drivers/infiniband/core/device.c */
static struct class ib_class = { /* InfiniBand device class */
  .name        = "infiniband",
  .dev_release = ib_device_release,
  .dev_uevent  = ib_device_uevent,
  .ns_type     = &net_ns_type_operations,
  .namespace   = net_namespace,
};
static noinline void rdma_init_coredev(struct ib_core_device *coredev,
  struct ib_device *dev, struct net *net) {
  ...
  coredev->dev.class = &ib_class; /* specify device class */
  ...
}
struct ib_device *_ib_alloc_device(size_t size) {
  ...
  rdma_init_coredev(&device->coredev, device, &init_net);
  ...
  return device;
}
int ib_register_device(struct ib_device *device, const char *name,
  struct device *dma_device) {
  ...
  ret = device_add(&device->dev);
  ...
}
static int __init ib_core_init(void) { /* InfiniBand class entry point */
  ...
  ret = class_register(&ib_class);
  ...
}
fs_initcall(ib_core_init);
```

(a) Part of InfiniBand class driver (`ib_core`) code.

```c
/* drivers/infiniband/hw/mlx5/main.c */
static int mlx5_ib_stage_ib_reg_init(struct mlx5_ib_dev *dev) {
  ...
  return ib_register_device(&dev->ib_dev, name, &dev->mdev->pdev->dev);
}
static int mlx5r_probe(struct auxiliary_device *adev,
  const struct auxiliary_device_id *id) {
  ...
  dev = ib_alloc_device(mlx5_ib_dev, ib_dev); /* dev struct w/ ib_class */
  ret = mlx5_ib_stage_ib_reg_init(dev); /* simplified */
  ...
}
static int __init mlx5_ib_init(void) { /* entry point */
  ...
  ret = auxiliary_driver_register(&mlx5r_driver);
  ...
}
module_init(mlx5_ib_init);

/* /include/rdma/ib_verbs.h */
#define ib_alloc_device(drv_struct, member) \
  container_of(_ib_alloc_device(sizeof(struct drv_struct) + ...))
```

(b) Part of InfiniBand device driver (`mlx5_ib`) code.

Figure 2: Device registration with the InfiniBand class.

image [84]. The platform device includes essential hardware devices (e.g., ACPI, PCI switch) and para-virtualization stacks (e.g., Hyper-V, KVM, and Xen) that must be initialized in advance to allow the kernel to load LKMs (from block storage), and legacy or System-on-Chip (SoC) hardware components without an enumerable bus for which the kernel cannot perform the driver binding.

Driver loading and symbol dependency. In Linux, an LKM (both device driver and software module) cannot be loaded into the kernel memory unless all its symbol dependencies are resolved [4]. In particular, an LKM can have a number of undefined symbols which implies that it depends on functions or global variables defined in the kernel image or other modules. For example, during initialization, device drivers will invoke bus-specific driver registration functions defined in the corresponding bus drivers. Thus, the kernel must load bus drivers into its memory before loading any other device drivers depending on them. While building the kernel, the build system figures out all such symbol dependencies between LKMs and creates a database file called `modules.dep`. Later, when the kernel or a privileged user is trying to load an LKM (e.g., using `modprobe`), `modules.dep` is inspected to load all ancestor modules before loading the target module.

3 THREAT MODEL AND ASSUMPTION

This paper considers a non-privileged adversary who aims to compromise the operating system kernel through exploiting its security vulnerabilities. In particular, the kernel might contain device drivers which have unknown or unpatched security vulnerabilities or are associated with other vulnerable kernel modules or functions. These vulnerable drivers, modules, or functions can be loaded into the kernel address space during the boot process because they are built into the kernel image or automatically loaded via the configuration or user-space activities [5, 16, 35, 66, 75]. The adversary can exploit them to eventually compromise the entire kernel.

We make the following assumptions. First, the operating system kernel can reliably probe the underlying hardware devices. For example, it can rely on BIOS/UEFI and ACPI as well as the hypervisor if the machine is virtualized to enumerate all hardware devices. This ground-truth information becomes our basis to distinguish operative device drivers (i.e., drivers with corresponding hardware devices) from inoperative device drivers (i.e., drivers without corresponding hardware devices). Second, we have access to the kernel source code (including device drivers and modules) and build configuration options of the target system images to debloat.

4 DESIGN

In this section, we describe the design of HACKSAW. HACKSAW aims to accurately reduce the kernel attack surface without false removals by identifying inoperative device drivers based on the hardware device inventory information and other kernel modules and kernel functions which tightly depend on the inoperative drivers.

Figure 3 shows an overview of HACKSAW, which consists of four major tasks:

- **T1.** Identify device drivers from the kernel source (§4.1)
- **T2.** Analyze the dependencies between the identified device drivers and other kernel components to construct a hardware-centric dependency graph (§4.2)
- **T3.** Figure out the device inventory of a target machine (§4.3)
- **T4.** Remove inoperative device drivers and kernel functions based on the dependency graph and device inventory (§4.4)

4.1 Device Driver Lookup (T1)

HACKSAW identifies device drivers from the Linux kernel source, which it can safely delete if no corresponding hardware device is attached to the machine. Device drivers have ID tables containing the hardware IDs to perform the device binding (§2). Therefore, we aim to identify ID tables contained in source code files, object files, and LKMs to identify device drivers.

Separate driver. Figuring out whether an LKM is a device driver is straightforward because, if it is, its metadata contains `modalias` which is converted from ID tables through the `MODULE_DEVICE_TABLE()`

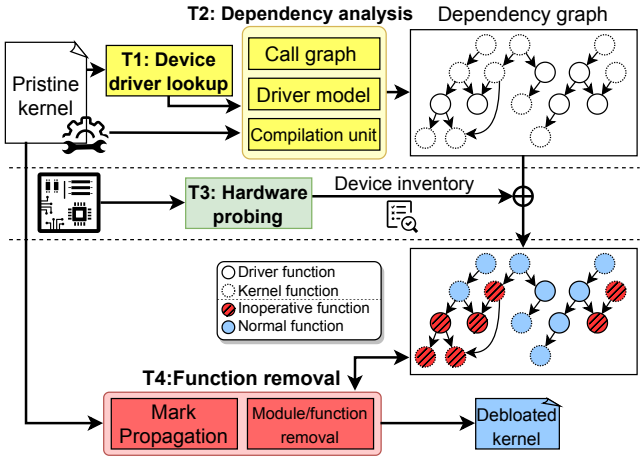

Figure 3: Overview of Hacksaw.

macro and the `depmod` command. Unfortunately, this technique does not work for built-in device drivers (e.g., for platform devices).

Built-in driver. We design a static-analysis algorithm to identify built-in device drivers and their hardware IDs from the kernel source. This algorithm is inspired by the device/driver registration procedure (§2). During the loading of a device driver, the kernel invokes the driver's entry-point function which will eventually invoke the custom device/driver registration functions (defined in its class or bus driver) whose arguments are data structures containing its hardware IDs (e.g., its ID table). Therefore, to detect built-in device drivers, our algorithm first finds entry-point functions and custom device/driver registration functions from the kernel source. Then, for each pair of entry-point function and custom registration function, it checks whether there is an explicit control flow (i.e., no indirect jump and call) from the entry-point function to the registration function along with its hardware IDs as an argument. If there is, the algorithm treats the function as a driver entry-point function and collects its hardware IDs.

Identify entry-point function: Identifying entry-point functions is straightforward because all entry-point functions are mapped to specific symbols to be invoked by the kernel. They will be mangled if they are embedded into the kernel image, but it is deterministic.

Identify custom device/driver registration: Our algorithm is based on two observations to accurately identify custom device and driver registration functions. First, as mentioned in §2, devices and drivers must be eventually registered with the kernel core through `device_register()`/`device_add()` and `driver_register()`, respectively. Therefore, every custom registration function will call them at the end. Second, these registration functions are expected to be invoked by external entities (i.e., device drivers), so they must be globally callable (i.e., their symbols are exported.) Based on these two observations, we detect global functions which eventually call `device_register()` and/or `driver_register()`.

Control flow and hardware ID: Once we identify entry-point and custom registration functions, we check the explicit control flows from entry-point functions to custom registration functions. While checking such a control flow, we perform a backward slicing starting from the registration function call to the entry-point function to extract the hardware IDs. The device/driver registration functions expect an argument for a device or driver structure containing hardware IDs. This data structure is well specified, so we check its field containing hardware ID to conduct the backward slicing.

However, this algorithm has a limitation: it only identifies the entry-point function of a built-in device driver. A device driver typically consists of multiple functions which span multiple source files. Thus, we need another mechanism to identify other functions belonging to the built-in driver to effectively delete it if it is inoperative. Our intuition to overcome this limitation is that, if we carefully analyze the dependencies between the entry-point function and other functions, we may be able to detect the driver's other functions. We will explain it in §4.2.

4.2 Dependency Analysis (T2)

Once Hacksaw identifies device drivers, it first constructs a directed dependency graph based on the identified drivers and then gradually populates the graph with other kernel functions and LKMs related to the identified drivers. Specifically, it conducts dependency analysis to detect other non-driver functions and LKMs that depend on the identified drivers or that the identified drivers depends on, and includes them to the graph. Hacksaw uses three dependency analysis approaches: call-graph analysis, driver-model analysis, and compilation-unit analysis. Hacksaw repeatedly runs these analyses until it no longer finds any new function and dependency. Hacksaw attempts to delete functions and LKMs only if they are included in the dependency graph at the end.

Dependency graph. Hacksaw constructs a hardware-centric dependency graph which is a directed graph between driver-related kernel functions. Each vertex represents a function or a set of functions (e.g., LKM, object file), and each edge represents directional dependencies between them. If vertex v_a depends on vertex v_b (e.g., a calls b, a imports a symbol that b exports), the graph has a directed edge from v_b to v_a ($v_a \leftarrow v_b$). A vertex has two attributes representing 1) whether it is from a driver and 2) whether it is inoperative. An edge has an attribute to specify whether it is strong (unconditional) or weak (conditional). For example, symbol import is always strong whereas a function call can be either strong or weak.

Call-graph analysis. Hacksaw figures out call- and symbol-based dependencies between functions in LKMs or the kernel image. The symbol dependencies between LKMs are already identified by the kernel build system (i.e., `modules.dep`). Thus, we focus on identifying the dependencies 1) between built-in kernel functions and 2) between functions in LKMs and built-in kernel functions.

Our call graph analysis is largely standard [74] except for the following polices to avoid false removals of indirectly or conditionally callable functions. First, it does not analyze (i.e., does not delete) functions whose addresses can be taken (potentially for indirect calls). Second, it only considers direct function invocations when it checks the dependency—i.e., it ignores any indirect calls via function pointers. Third, it distinguishes conditional calls from unconditional calls and maintains it in each edge attribute.

We start our call-graph analysis from the entry-point functions of the identified built-in drivers and all functions in LKM device drivers while ignoring calls between LKMs (which are already identified). Whenever we find new functions during the analysis, we run the analysis against the new functions recursively until we no longer observe any new function and dependency.

Although our call-graph analysis is accurate and general, it relies on direct function calls so that it has no way to observe indirect dependencies potentially through indirect function calls and shared data. Thus, we consider two other approaches which can accurately handle specific indirect dependencies.

Driver-model analysis. The driver-model analysis is based on the semantics we explained in §2—device drivers are managed by their bus drivers according to their hardware buses and their class drivers according to their device classes. Therefore, we aim to identify bus and class drivers in order to detect other kernel functions which are related to device drivers through the bus and/or class drivers.

The driver-model analysis begins with identifying bus and class drivers based on their three features. First, these drivers invoke `bus_register()` and `class_register()` during initialization, respectively. Thus, we check the call graphs of LKMs and built-in kernel functions to confirm whether they (eventually) invoke these functions. Second, they have functions to assign a driver or device to a certain bus or device class by storing a specific bus-type or device-class value to the corresponding field of a driver or device structure. Third, they define custom device/driver registration functions that we explained in §4.1. Once we identify the bus and class drivers, we include them into the dependency graph and run the call-graph analysis against them to extend the graph.

Compilation-unit analysis. The compilation-unit analysis identifies the compilation dependencies between drivers and kernel functions—i.e., whether they must be built together and are potentially linked into the same archive or object file. It is effective in discovering non-entry-point functions of built-in device drivers. Specifically, we identify minimal build configuration options that individual drivers or kernel functions depend on and check whether their build options cover those of other kernel functions or drivers. To this end, we first analyze the kernel build system (i.e., Kbuild Makefile) to identify minimal build configuration options to selectively compile the source files containing our target functions (e.g., entry-point function and device/driver registration function) and represent them as Boolean expressions (e_t). Second, we identify a set of Boolean variable assignments which satisfy e_t. Third, we identify other source files that will be built according to the assignment set (i.e., if a Boolean expression of their build configuration options is e_o, e_t implies e_o). For example, the kernel build system builds `ath3k.o` if both `CONFIG_BT` and `CONFIG_BT_ATH3K` are enabled (Figure 4). This build option further builds `bluetooth.o` which only requires `CONFIG_BT`. We need a Boolean Satisfiability (SAT) solver [19] to compare the satisfiability of their build configuration options. Finally, we include the functions contained in the same compilation-unit into the dependency graph if at lease one of them is already in the graph and run the call-graph analysis against them to extend the graph.

```
# drivers/Makefile
obj-$(CONFIG_BT) += bluetooth/
# drivers/bluetooth/Makefile
obj-$(CONFIG_BT_ATH3K) += ath3k.o
# And(Bool(CONFIG_BT), Bool(CONFIG_BT_ATH3K))

# net/Makefile
obj-$(CONFIG_BT) += bluetooth/
# net/bluetooth/Makefile
obj-$(CONFIG_BT) += bluetooth.o
# Bool(CONFIG_BT))

# And(Bool(CONFIG_BT), Bool(CONFIG_BT_ATH3K)) -> Bool(CONFIG_BT))
```

Figure 4: Part of Makefiles to build `ath3k.o` and `bluetooth.o`.

4.3 Hardware Probing (T3)

Device inventory. HACKSAW identifies a list of hardware devices attached to the target machine. This device inventory is necessary to distinguish inoperative device drivers from operative device drivers in the dependency graph (§4.2). To identify the set of devices on the target machine, HACKSAW first boots a minimal operating system—which can be based on the target kernel image it has to debloat but with `initramfs`—on the target machine. Then, it runs a probing script to collect all hardware-related information that the operating system kernel exposes (e.g., through `sysfs`) and, finally, compiles the device inventory. It is worth noting that this device inventory creation is the only procedure to be performed online.

Marking. Once the device inventory is prepared, HACKSAW traverses the dependency graph to locate the vertices (i.e., functions or modules) belonging to device drivers and mark them as "inoperative" only if their hardware IDs do not match any hardware device in the device inventory. HACKSAW does not falsely mark any vertices that do not have hardware IDs (i.e., they belong to software components).

4.4 Function Removal (T4)

Once the dependency graph (§4.2) is marked by the device inventory (§4.3), HACKSAW identifies all inoperative device drivers and kernel functions by propagating marks throughout the graph, and then finally deletes the corresponding LKM files or patches out the corresponding functions in the kernel image or LKMs.

Mark propagation. HACKSAW propagates "inoperative" marks throughout the dependency graph based on three rules. All these rules are repeatedly applied until no more vertex is markable. In addition, HACKSAW never marks nodes whose hardware IDs match any entry of the device inventory.

Rule 1. Descendant marking: If a vertex is a strong descendant of an already marked vertex (e.g., if a function unconditionally calls a marked function), HACKSAW marks the vertex.

Rule 2. Ancestor marking: If all strong or weak descendants of a vertex are marked (e.g., if all functions which conditionally or unconditionally call a function are marked), HACKSAW marks the vertex.

Rule 3. Bus/class ancestor marking: If a vertex is for a bus or class driver and all its strong or weak descendant vertices for device drivers (i.e., which can call its custom registration functions) are marked, HACKSAW marks the vertex. This rule is a relaxed variant of Rule 2 which is based on bus/class contexts: if no hardware device

is registered with a certain bus or class, the bus or class does not properly operate and is removable.

Module/function removal. Once "inoperative" marks are fully propagated, HACKSAW deletes the corresponding LKMs from the target system image (e.g., its root file system) and the corresponding functions embedded in the kernel image or LKMs. We consider two approaches to delete embedded functions: use a compiler pass to delete the functions during compilation or use a binary rewriting tool to patch out them. The former reduces the kernel memory footprint but requires kernel re-compilation. The latter requires no kernel re-compilation but wastes the kernel memory. Since reducing the memory footprint is not our main goal, we choose the latter for our implementation (§5), but the former also works.

5 IMPLEMENTATION

In this section, we explain a prototype implementation of HACKSAW for Linux-based systems. Our implementation for identifying device drivers and building the dependency graph consists of 1,000 lines of Python code (3.10) and 1,400 lines of C++ code for LLVM-based static analysis (LLVM 15.0 [48]). We use Whole Program LLVM (WLLVM [70]) to build the Linux kernel with our passes. Also, our debloating code consists of 1,420 lines of Shell Script and 150 lines of Python code. We use the Linux kernel version 5.19 as our baseline kernel—function names and numbers mentioned in this section are based on this version. Also, we focus on x86, so our results do not cover device drivers for other architectures like Arm and MIPS.

5.1 Device Driver Lookup

Separate drivers. As explained in §4.1, all LKMs which are device drivers contain `modalias` in their metadata section. Thus, we compile our Linux kernel (version 5.19) with the `allmodconfig` build option—which compiles drivers and modules as LKMs if possible—and run `modinfo` against all LKMs (i.e., `*.ko`) to collect their hardware IDs. In total, we observe 5,878 LKM device drivers (for x86).

Built-in drivers. To deal with built-in device drivers, we first figure out entry-point or initialization functions in the kernel. A device driver either defines its entry-point function or uses a pre-defined macro to automatically generate one. A device driver can name its own entry-point function arbitrarily (e.g., `mlx5_ib_init()`, `usb_init()`, `ib_core_init()`) and use a kernel-provided macro (e.g., `module_init()`, `subsys_initcall()`, `fs_initcall()`) to expose its entry-point function with deterministic mangling rules with certain priority of loading (Figure 1a, Figure 2). For example, Linux kernel version 5.12+ mangles every entry-point function in the format of `__initcall__kmod_<name>__<cnt>_<line>_<entryfn>_<lvl>` with macros, where `name` is a module name, `cnt` is a global macro counter, `line` is a line number in the source code, `entryfn` is an entry-point function name, and `lvl` is a pre-defined priority level. If a driver uses a pre-defined macro (e.g., `module_pci_driver()`, `module_usb_driver()`) to generate an entry-point function (Figure 1b), the macro names the entry-point function with a symbol which is a concatenation of a driver struct name and `_init`. We leverage these two naming patterns to identify entry-point functions in the kernel image. In total, we find 6,358 entry-point functions 1,174 of which belong to built-in-only drivers or modules.

Next, we figure out custom device and driver registration functions. We build call graphs starting from every global function and check whether their vertices contain `device_register()`, `device_add()`, or `driver_register()`. If they do, we treat the global function as a custom registration function. In total, we find 186 custom device registration functions and 74 custom driver registration functions.

Finally, we conduct backward slicing against the device/driver structure arguments of the custom registration functions whose field has an ID table. This backward slicing ends up with global ID table structure variables and we extract hardware IDs from them. We cross-check the type of the ID table with the device ID structs defined in the Linux kernel source (e.g., `file2alias.c`). HACKSAW currently supports 12 device ID structures including `pci_device_id`, `usb_device_id`, `x86_cpu_id`, and `hv_vmbus_device_id`, which covers 98.8% of all device drivers according to `modules.alias`. Note that there is no technical challenge in handling all device ID structures—partial support is only an engineering decision.

5.2 Dependency Analysis

Dependency graph. In our prototype implementation, we separate dependency graphs (§4.2) for LKMs and kernel functions to address their different levels of dependency in a simplified manner. In the LKM-level dependency graph, every vertex represents an LKM and every directed edge represents the symbol import/export—an LKM importing a symbol is a child and an LKM exporting a symbol is a parent. To this end, we check each LKM's symbol table (using `nm`) and `modules.dep`. In the function-level dependency graph, every vertex represents a kernel function and every directed edge represents a function call—a caller is a child and a callee is a parent. We additionally maintain whether each function resides in the kernel image or an LKM to ease the later function removal (§5.4).

Call-graph analysis. We build function call graphs on top of LLVM IR over the whole Linux kernel and LKMs. For every function, we compute two lists each including its callers and callees. We use `modules.dep` to identify symbol-level dependencies between LKMs.

Driver-model analysis. We identify bus and class drivers to perform the driver-model analysis. As explained in §4.2, the bus and class drivers 1) invoke `bus_register()` and `class_register()`, 2) assign bus-type and device-class variables to driver and device structures, and 3) define custom driver and device registration functions, respectively. The first one is straightforward—we conduct a call-graph analysis to identify which LKMs and built-in kernel functions invoke them. Also, the third one is already done because we identify a list of custom driver and device registration functions for the driver lookup (§5.1). To deal with 2), we write an LLVM pass and a source-code text parser. In particular, we first locate all bus type (`struct bus_type`) and device class (`struct class`) definitions and collect their variable names (e.g., Line 2 in Figure 1a and Figure 2a). Then, we look for functions that assign the collected variable names to the corresponding field of driver and device structures (e.g., Line 11 in Figure 1a and Line 12 in Figure 2a). Finally, we seek which LKMs and built-in object files define these functions. In total, we find 90 bus types and 72 device classes.

Compilation-unit analysis. To conduct the compilation-unit analysis, we 1) identify minimal build configuration options for

compiling each kernel source file, 2) figure out configuration variable (Boolean) assignments to build a certain source file (e.g., which defines entry-point or custom registration functions), and 3) seek other source files that will be built as well due to the configuration variable assignments. First, we use kmax [28] which returns minimal build configuration options for building each kernel source file as a Boolean expression—i.e., a sequence of configuration options along with `And`, `Or`, and `Not` operators. Then, to figure out a set of variable assignments which satisfies the Boolean expression for a source file, we successively assign a `True` or `False` value to its variables and check its satisfiability using Z3 [19] for all possible assignment combinations. Instead of testing all possible combinations (which is computationally expensive), we can extract actual assignments from the build configuration option of a target system image. Lastly, we apply the set of satisfying variable assignments against the Boolean expression for building each source file with Z3 to confirm whether the assignment is satisfied. If a variable assignment satisfies the Boolean expressions of multiple source files, these files belong to the same compilation unit.

5.3 Hardware Probing

Device inventory. Our prototype implementation relies on a Linux installation to probe hardware information. The Linux installation running in our target machine enumerates all hardware components and exposes them via `sysfs` (i.e., `/sys/devices`). Then, we run a script to dump all `modalias` files [71] inside `sysfs` which contains all hardware IDs of the platform. We use a standard installation of Ubuntu Server 22.04 which is slightly modified to automatically runs our script for this purpose.

Marking. We compare the hardware IDs we extracted from the device drivers against the device inventory. The hardware ID from a device driver often contains wildcard characters (e.g., `*` matches any character of any length) to be compatible with minor revisions in the hardware devices. Thus, we convert the wildcard expressions to a regular expression and match it against the device inventory.

5.4 Function Removal

Mark propagation. We associate a device's hardware ID with 1) its separate device driver (i.e., LKM) or 2) its entry-point function or compilation unit if it is a built-in driver. We mark the device driver or the compilation unit as "inoperative" when we confirm that the target hardware device inventory does not have a match with the hardware ID. We implement Rule 1, 2, and 3 in a Python script as described in §4.4 to traverse the dependency graph and propagate the "inoperative" mark.

Module removal. To remove LKMs of inoperative device drivers or kernel components, we mount the target system image and then remove inoperative LKMs stored in `/lib/modules/<kernel version>`. Also, we update `modules.dep` accordingly to avoid meaningless module loading attempts by system services.

Initial RAM file system: In addition to removing the LKMs in the root file system, we remove the LKMs contained in the initial RAM file system (`initramfs`). Our first step is to unpack `initramfs`. The standard format for `initramfs` is CPIO archive, but different Linux distributions pack their `initramfs` differently—e.g., Ubuntu packs three CPIO archives into a single `initramfs` file, including two for AMD and Intel CPU microcode binaries followed with one compressed CPIO archive with its minimal root file system. In our implementation, we infer the `initramfs` layout by recursively unpacking it with CPIO format and cutting off the part that successfully unpacked, until we have reached a compressed binary format (e.g., zstd, gzip). After the unpacking procedure, we decompress `initramfs` and remove the inoperative drivers and modules from it. In the end, we repack `initramfs` by compressing the patched file system and putting all CPIO archive parts back together.

Function removal. Once we identify inoperative functions in the kernel image and LKMs, we remove them from the corresponding binary files. We use Capstone [12] for this. We first locate where the inoperative functions are. If an inoperative function resides in the kernel image, we use the Linux kernel's system-wide symbol table (i.e., `System.map`) to identify its offset. If an inoperative function resides in an LKM, we check its ELF header information to calculate the function offset. After we locate the inoperative function, we patch it based on linear disassembly according to its types. If it is a built-in entry-point function, the kernel invokes it regardless of whether the corresponding hardware device exists. However, the kernel does not check its return value, so we simply replace the function with a direct return. If it is not an entry-point function, the kernel must not invoke it according to our dependency analysis. Therefore, we replace its function body with the `nop` instruction. In addition, we identify and skip the `Ftrace` stub—for dynamic tracing—at the beginning of functions because patching them out can result in system failures.

Unpacking and repacking the kernel: The Linux kernel image is shipped in compressed format as `bzImage` consisting of the compressed kernel binary and the bootstrap loader. Therefore, to patch out the inoperative functions in the kernel image, we need to unpack `bzImage` first to extract the raw image and then repack it back after the patching. We use the `extract-vmlinux` script contained in the Linux source code to unpack `bzImage`. Repacking the kernel is challenging because the patched kernel is not recompressed back to exactly the same size and layout as it was before the patching—it is no longer compatible with the bootstrap loader. To solve this problem, we pre-build another instance of the Linux kernel with the same kernel version and configuration to collect setup binaries, offset information (i.e., `zoffset.h`), and build scripts. We use them to correctly repack the patched kernel image.

Decompressing and recompressing LKM: Some Linux distributions (e.g., CentOS, openSUSE) compress their LKMs to save disk space. Thus, during the patching of LKMs, we decompress and recompress them to comply with the system settings.

6 EVALUATION

In this section, we evaluate HACKSAW by answering the following questions:

- **RQ1.** How many inoperative drivers, modules, and functions does HACKSAW delete? (§6.2)
- **RQ2.** How many security vulnerabilities does HACKSAW potentially mitigate? (§6.3)
- **RQ3.** Is the result of HACKSAW valid? (§6.4)
- **RQ4.** Does HACKSAW ensure compatibility? (§6.5)

Table 1: Hardware platforms and their components.

Type	AWS	AZ1	AZ2A	AZ2I	GCP	KVM	HV
	t2.micro	D2s-v3	D2as-v5	D2s-v5	e2-micro	Default	
ACPI	18	20	9	9	13	17	20
CPU	1	1	1	1	1	1	1
HID	0	1	1	1	0	0	1
PCI	6	6	0	1	7	6	5
SCSI	0	1	1	1	1	2	2
SerIO	2	2	1	1	2	2	2
Virtio	0	0	0	0	4	0	0
VMBus	0	13	10	13	0	0	15
XenBus	2	0	0	0	0	0	0

* **AWS**: AWS VM (Intel)
* **AZ1**: Azure Gen1 VM (Intel)
* **AZ2A**: Azure Gen2 VM (AMD)
* **AZ2I**: Azure Gen2 VM (Intel)
* **GCP**: GCP VM (Intel)
* **KVM**: QEMU-KVM (Intel)
* **HV**: Windows Hyper-V Gen1 VM (Intel)

Table 2: System images with their modules and built-in kernel functions.

Image	Version		Module		Kernel function		
	Distro	Kernel	Total	Driver	Total	Entry	DrvEntry
Bare-metal							
Debian	11	5.10	3,899	1,454	43,712	597	48
Fedora Server	36	5.17	3,893	1,430	61,107	797	73
Fedora Workstation	36	5.17	4,015	1,451	61,107	797	73
Ubuntu Server	22.04	5.15	6,040	2,296	65,489	810	111
openSUSE Leap	15.4	5.14	4,604	1,681	48,130	675	47
Cloud image							
CentOS (AWS)	9	5.14	2,152	664	53,638	714	53
Debian (Generic)	11	5.10	937	96	40,984	505	19
Fedora Cloud Base (GCP)	36	5.17	1,859	620	61,107	797	73
Ubuntu Server (AWS)	22.04	5.15	1,000	144	65,710	809	111
Ubuntu Server (Azure)	22.04	5.15	884	82	63,027	726	76
Ubuntu Server (GCP)	22.04	5.15	998	145	66,375	822	112
openSUSE Leap (GCP)	15.4	5.14	2,227	635	48,270	676	47

* **Entry**: Entry-point functions
* **DrvEntry**: Entry-point functions with device IDs

6.1 Environment

Hardware platform. As target hardware platforms, we choose virtual machine instances of the three public cloud services (Amazon AWS EC2, Microsoft Azure, and Google Cloud Platform (GCP)) and two desktop virtualization platforms: QEMU-KVM and Windows Hyper-V. All these have predictable, different hardware configurations provided by different virtualization technologies. Specifically, the AWS EC2 instance we choose uses Xen, so its guest machines are expected to support XenBus devices. Both Azure and Windows Hyper-V use Hyper-V, so its guest machines are expected to support Virtual Machine Bus (VMBus). GCP uses KVM, so its guest machines are expected to support Virtio devices. Despite all the differences among these virtualization platforms, they commonly attach popular types of devices to their guest machines including ACPI, CPU, PCI, SCSI, and serial I/O devices. This makes our selection of cloud platforms generic enough for our evaluation.

Overall, we end up with seven different hardware device inventories in total. This includes three different hardware configurations for Microsoft Azure to evaluate configuration differences for the same cloud provider: Generation 1 (BIOS) VM and Generation 2 (UEFI) VM with AMD or Intel CPU (Table 1).

System images. We prepare various system images based on popular Linux distributions including CentOS, Debian, Fedora, openSUSE, and Ubuntu. We create five bare-metal system images using their Optical Disc (ISO) images. Also, we download or dump their cloud image versions (seven in total) specialized for AWS, Azure, and GCP (Table 2). Particularly for Ubuntu, we prepare the cloud images for the three cloud services.

Evaluation machine. We use a desktop computer featuring an Intel Core i9-12900K CPU and 64 GiB of RAM for all our evaluations.

6.2 Module and Function Removal (RQ1)

We evaluate how many inoperative device drivers and kernel modules as well as kernel functions Hacksaw can remove from our system images when we debloat them for our target platforms. Before explaining the removal results, we first check how many LKMs and kernel functions (functions embedded in the kernel images) exist in our system images (Table 2). As expected, the cloud images have fewer modules (and device drivers) in total than the bare-metal images because the cloud images are already specialized for their target platforms [45]. Interestingly, the differences between the bare-metal and cloud images in terms of kernel functions are marginal. This is potentially because built-in functions are critical to system execution such that cloud image maintainers do not try to specialize them. Note that, in Table 2, we determine the number of device drivers by comparing all LKMs and built-in entry-point functions against our device driver lookup results. Since our prototype does not cover all device IDs (it covers 98.8% of all device IDs §5.1,) the number of device drivers enumerated here might be slightly smaller than the actual number, but any such difference would be negligible.

Bare-metal and cloud image reduction. Figure 5 shows the number of device drivers, kernel modules, and kernel functions that Hacksaw removes from or maintains in our system images based on the device inventory of our target platforms. In particular, we break down the number in terms of why we delete or preserve them: 1) inoperative device drivers (**driver**) or their entry-point functions if they are built-in drivers (**drvEntry**), 2) modules or functions depending on the inoperative drivers (**dependent**), 3) modules whose functions are partially removed (**patched**), and 4) the remaining drivers, modules, or functions (**remaining**). Both driver- and dependent-based removals significantly affect the overall reduction ratio. Hacksaw removes 61.4%–69.7% of LKMs from the bare-metal system images and 11.8%–55.5% of LKMs from the cloud images. Also, it removes 30.4%–36.4% of functions from the kernel images in the bare-metal images and 25.6%–35.7% of functions from the kernel images in the cloud images. Therefore, we conclude that Hacksaw effectively reduces the attack surface of both bare-metal and cloud images even though the cloud images are already specialized by their maintainers.

Distinct reduction. We check whether and how different device inventories affect Hacksaw's reduction of the same system image. We choose three debloated images for AWS, AZ2I, and GCP which are based on the bare-metal Ubuntu Server image, and then compare the removed kernel modules and functions between them. Table 3 shows that, even though they are based on the same image,

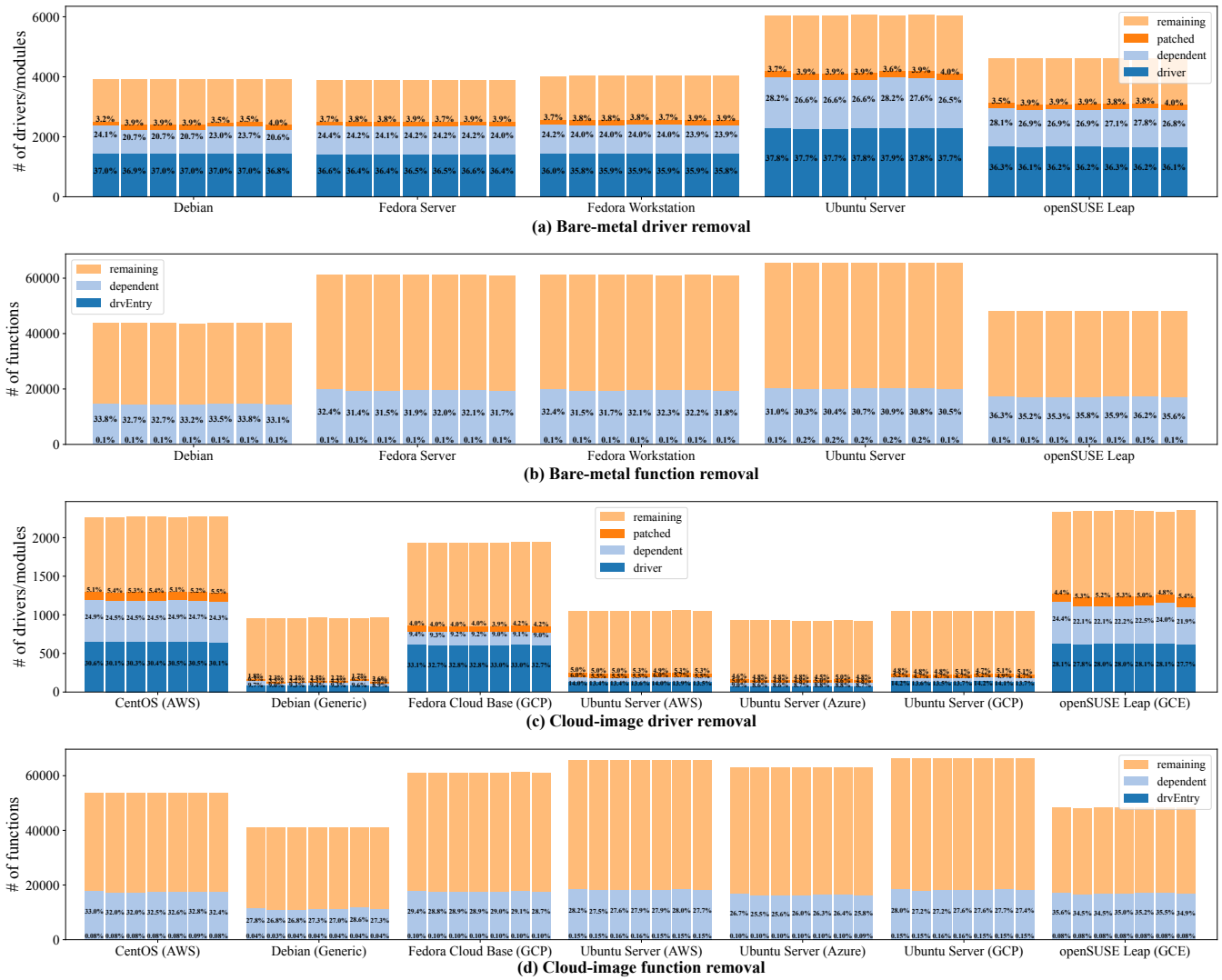

Figure 5: The number of drivers/modules that HACKSAW removes from or maintains in each combination of the system images and the target platforms (left to right: AWS, AZ1, AZ2A, AZ2I, GCP, KVM, and HV).

Table 3: Comparison of removed modules and functions for Ubuntu Server with different profiles (AWS, AZ2I, GCP).

Difference	Modules	Kernel functions
AWS − GCP	3	94
AWS − AZ2I	7	140
GCP − AWS	6	240
GCP − AZ2I	9	122
AZ2I − AWS	16	420
AZ2I − GCP	15	256

the removed modules and functions are distinct. For example, the debloated AWS image has no Virtio and VMBus drivers whereas the debloated GCP image has no VMBus and XenBus drivers. Also, the debloated AWS image maintains a few extra ACPI and PCI devices like `pata_acpi` whereas both AZ2I and GCP images do not.

`initramfs` reduction. We also check whether HACKSAW reduces the attack surface of `initramfs` by removing inoperative LKMs it contains. Table 4 shows that HACKSAW removes 38.8%–61.1% of LKMs contained in the bare-metal system images and 23.8%–46.4% of LKMs contained in the cloud images. Thus, HACKSAW effectively reduces the attack surface of `initramfs`.

6.3 Vulnerability Mitigation (RQ2)

We evaluate how effective HACKSAW is in terms of potential vulnerability mitigation. To this end, we analyze CVEs which are historically related to Linux kernel drivers or modules and check how many driver/module CVEs HACKSAW can remove based on hardware absence. We first collect all Linux kernel CVEs since 2015

Table 4: Reduction of LKMs in `initramfs`.

Image	Removed	Total	%
Bare-metal			
Debian	412	674	61.1
Fedora Server	118	303	38.9
Fedora Workstation	118	303	38.9
Ubuntu Server	792	1,533	51.7
openSUSE Leap	157	405	38.8
Cloud image			
CentOS (AWS)	289	649	44.5
Debian (Generic)	37	124	29.8
Fedora Cloud Base (GCP)	216	571	37.8
Ubuntu Server (AWS)	91	335	27.2
Ubuntu Server (Azure)	70	294	23.8
Ubuntu Server (GCP)	87	330	26.4
openSUSE Leap (GCE)	389	839	46.4

Table 5: Decomposition of CVE mitigations on CVSS scores.

Image	Removed/Total			
	Low	Medium	High	Critical
Bare-metal				
Debian	10/24	230/459	191/368	12/25
Fedora Server	8/22	214/349	187/321	12/21
Fedora Workstation	9/25	220/384	189/338	12/24
Ubuntu Server	10/26	257/438	222/387	14/27
openSUSE Leap	11/25	264/438	210/375	13/25
Cloud image				
CentOS (AWS)	5/16	164/289	140/252	7/15
Debian (Generic)	2/12	42/185	42/171	1/10
Fedora Cloud Base (GCP)	1/11	66/186	48/169	4/11
Ubuntu Server (AWS)	2/17	60/216	47/189	3/14
Ubuntu Server (Azure)	1/16	35/193	29/179	1/12
Ubuntu Server (GCP)	0/14	56/210	43/183	3/14
openSUSE Leap (GCE)	7/18	208/370	164/309	10/20

which are bundled with patch commits [49]. Then, we match the patch commits with the kernel drivers and modules based on their source code file paths in order to associate the CVEs with the drivers or modules. In total, we find 1,644 CVEs related to drivers or modules.

With the total set of driver/module CVEs, we check how many driver/module CVEs our system images individually contain and how many of them HACKSAW can potentially mitigate by removing their inoperative drivers and modules. Figure 6 shows the results. Overall, HACKSAW is able to mitigate 10.7%–57.4% of the driver/module CVEs contained in the system images. We further check the severity of driver/module CVEs mitigated by HACKSAW based on their Common Vulnerability Scoring System (CVSS) [22] scores. As shown in Table 5, HACKSAW mitigates CVEs with high or critical CVSS scores as well. Thus, we conclude that HACKSAW is effective at mitigating kernel security vulnerabilities. We further discuss whether these driver/module CVEs can be triggered without corresponding hardware components in §7.

6.4 Validity (RQ3)

We evaluate whether the outcome of HACKSAW is valid. That is, we aim to check whether HACKSAW might falsely remove device drivers which are needed. This evaluation requires ground truth for comparison, but, unfortunately, there is no complete ground truth we can rely on. A potential candidate we considered is LKDDb [14], but we observed that it has many false positives because it relies on text matching and heuristics. Instead, we use the cloud images which are manually trimmed by experts to be launched in certain cloud or virtualization environments [45]. We emphasize that the cloud images are not fully minimized for their target platforms (as shown in §6.2). They typically contain some redundant drivers, which allow them to be migrated between different public cloud services [32, 33, 68]. Thus, our validity evaluation still requires best-effort manual verification.

Table 6: Comparison of the number of removed drivers by HACKSAW and the cloud images.

Image	Platform	Removed				Total
		HACKSAW	Cloud	$\|H - C\|$	$\|C - H\|$	
Debian	KVM	1,448	1,358	94	3	1,454
Fedora Server	GCP	1,424	810	614	0	1,430
Fedora Workstation	GCP	1,445	831	614	0	1,451
Ubuntu Server	AWS	2,289	2,149	142	2	2,296
Ubuntu Server	AZ2I	2,287	2,204	84	0	2,296
Ubuntu Server	GCP	2,290	2,147	144	1	2,296
openSUSE Leap	GCP	1,674	1,045	629	0	1,681

We debloat our five bare-metal images according to the cloud images we collected. For example, since we have the Debian cloud image for KVM (generic), we debloat the bare-metal Debian image for KVM. Also, since we have the Ubuntu cloud images for AWS, Azure, and GCP, we debloat the bare-metal Ubuntu Server image for each of them. Then, we compare the debloated bare-metal images against the corresponding cloud images. Table 6 shows the results. In total, the system images debloated by HACKSAW have 3.6%–42.9% fewer device drivers than the cloud images. We compare the hardware IDs of the drivers deleted by HACKSAW against the hardware device inventory (based on `sysfs`) of each platform. We confirm that there is no false removal. In particular, we manually verify the device/bus type and their corresponding hardware IDs for each driver we removed. For all the extra drivers HACKSAW removes, we verify and exclude bus types that do not exist in our target platforms including Plug-n-Play, USB, Type-C, and I3C. We also check the device types which are exclusive to specific cloud platforms. For example, Azure uses Hyper-V, so it does not support XenBus and Virtio drivers. For the remaining device drivers (ACPI, CPU, I2C, MDIO, PCI, and platform devices), we check the hardware IDs they embed and confirm that these devices do not exist in the target cloud platforms.

In addition, we observe five drivers in total which are deleted by some cloud images but not HACKSAW: `e1000`, `fjes`, `floppy`, `parport_pc`, and `sb_edac`. Since their corresponding hardware devices exist in the target platforms, we believe the cloud image maintainers have deleted them for other reasons (e.g., they might be barely used.)

6.5 Compatibility (RQ4)

Lastly, we evaluate whether HACKSAW is compatible with popular applications and maintains the system stability. First, we choose seven real-world applications—7zip, Memcached, NGINX, Octave,

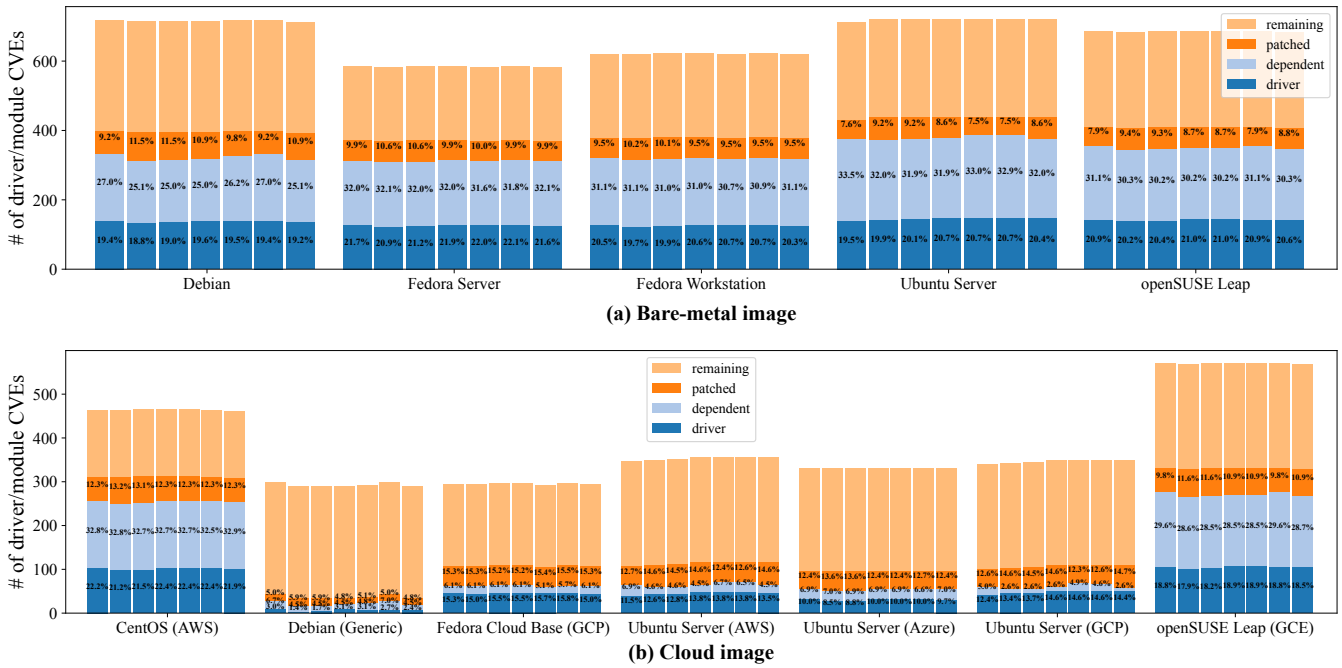

Figure 6: The number of driver/module CVEs that HACKSAW reduces from each combination of the system images and the target platforms (left to right: AWS, AZ1, AZ2A, AZ2I, GCP, KVM, and HV).

Table 7: Phoronix Test Suite results (KVM).

Image	7zip	Memcached	NGINX	Octave	OpenSSL	Redis	SQLite
Bare-metal							
Debian	✔	✔	✔	✔	✔	✔	✔
Fedora Server	✔	✔	✔	✔	✔	✔	✔
Fedora Workstation	✔	✔	✔	✔	✔	✔	✔
Ubuntu Server	✔	✔	✔	✔	✔	✔	✔
openSUSE Leap	✔	✔	✔	✔	✔	✔	✔
Cloud image							
CentOS (AWS)	✔	✔	✔	✔	✔	✔	✔
Debian (Generic)	✔	✔	✔	✔	✔	✔	✔
Fedora Cloud Base (GCP)	✔	✔	✔	✔	✔	✔	✔
Ubuntu Server (Azure)	✔	✔	✔	✔	✔	✔	✔
Ubuntu Server (AWS)	✔	✔	✔	✔	✔	✔	✔
Ubuntu Server (GCP)	✔	✔	✔	✔	✔	✔	✔
openSUSE Leap (GCP)	✔	✔	✔	✔	✔	✔	✔

OpenSSL, Redis, and SQLite—from the Phoronix Test Suite [64] which run with the 12 original system images on KVM without problems. Then, we run them with the 12 images patched by HACKSAW for KVM (Table 7). All test runs succeed with negligible performance difference. Thus, we conclude that HACKSAW is compatible with real-world applications.

Next, to check system stability, we run the LTP testsuite [50] against all our system images before and after applying HACKSAW to them. We run LTP against the original system images to filter out tests that do not work even for them. Among the 2116 tests of LTP, 2037–2107 tests work with the original images. Then, we run those successful tests against the patched images. We confirm that 2036–2107 tests work with the patched images. We carefully investigate all LTP failures (12 in total) and confirm that they fail because their failure check routines are incompatible with HACKSAW. For example, LTP testcases "cve-2017-1000380" and "cve-2018-7566" check whether a sound device driver is loaded before performing actual tests (i.e., attack the driver). If they fail to detect a loaded sound driver, they simply terminate with a failure even though they cannot attack the driver. Since HACKSAW passes all LTP testcases except for such manually confirmed incompatible ones, we conclude that HACKSAW does not introduce stability problems.

7 DISCUSSION

Vulnerabilities triggerable without hardware. HACKSAW mitigates security vulnerabilities potentially residing in kernel drivers, modules, and functions by removing them if their corresponding hardware components are not attached to a target machine. However, if all these vulnerabilities cannot be triggered without corresponding hardware devices, HACKSAW's security advantages might be shallow because they are no longer directly exploitable. To quantify the security impact of HACKSAW, we study certain driver/module CVEs mitigated by HACKSAW in depth to confirm whether they might be triggered without corresponding hardware. Since there is no public database for this, we first collect and investigate 189 Linux kernel CVEs which have public PoCs, write-ups, or fuzzer harnesses from various sources[1]. Among them, 73 CVEs are related to drivers or modules contained in our system images and 39 out of these driver/module CVEs are inoperative and thus removable by HACKSAW. We manually verify each one of them. In total, we identify 29 driver/module CVEs which can be potentially

[1]https://github.com/xairy/linux-kernel-exploitation, https://github.com/SecWiki/linux-kernel-exploits, https://github.com/PurpleVsGreen/beacown, https://github.com/tg12/PoC_CVEs

Table 8: CVEs which are potentially triggered without corresponding hardware and can be mitigated by HACKSAW.

Channel	CVEs
Sound-core	CVE-2016-2543, CVE-2016-2544, CVE-2016-2545, CVE-2016-2546, CVE-2016-2547, CVE-2016-2548, CVE-2016-2549, CVE-2017-1000380
USB/IP	CVE-2016-3955, CVE-2016-4482, CVE-2021-39685, CVE-2022-25375
InfiniBand/iSCSI	CVE-2016-4565, CVE-2021-27363, CVE-2021-27364, CVE-2021-27365
mac80211_hwsim	CVE-2018-8087, CVE-2022-41674, CVE-2022-42719, CVE-2022-42720, CVE-2022-42721, CVE-2022-42722
Netlink	CVE-2016-8658, CVE-2019-12984, CVE-2021-23134
Fake Bluetooth	CVE-2021-3573
Virtual CAN	CVE-2021-3609, CVE-2021-32606
VSOCK	CVE-2021-26708

triggered without hardware—i.e., they can be potentially triggered with software components or virtual devices included in the Linux kernel (not in a hypervisor) as shown in Table 8. It is worth noting that this study is limited because it is based on manual analysis of CVE and PoC descriptions. Figuring out whether certain vulnerabilities are triggable and exploitable is still an open research question.

Other architectures. Our HACKSAW prototype currently only supports x86 (x64), but it can be easily extended to support other architectures. This is because HACKSAW relies mostly on source code and the ELF specification which architecture independent. Also, many server- and desktop-class machines in other architectures (e.g., Arm) already support both UEFI and ACPI [6], so we can reliably probe and profile their hardware platforms. Only the binary patching part needs to to be adjusted due to different instruction sets, but it is straightforward because our implementation is based on Capstone [12] which supports multiple architectures.

Limitations. In HACKSAW, we keep our analysis conservative to avoid false positives. To this end, it might fail to delete certain drivers, modules, or kernel functions even though they are safe to delete. We conclude that the limitations of HACKSAW are coming from three parts. First, HACKSAW selects the initial set of removable (marked) modules and functions based on hardware information, not based on whether they will be used or invoked by others (§4.3). Thus, if it fails to extract the hardware information from certain modules and functions (e.g., for legacy hardware without IDs), it does not delete them regardless of their usage. Second, HACKSAW deletes a module or a function only if it unconditionally depends on or invokes marked modules or functions (§4.2 and §4.4). Certain conditional dependencies or function invocations will not be resolved in practice, but HACKSAW does not leverage them. Third, HACKSAW does not delete functions if their addresses are taken for potential indirect calls (§4.2). Thus, it does not delete unused functions if they are registered with function tables. To overcome these limitations, HACKSAW can adopt advanced analysis techniques [51], but it should accept some false positives.

8 RELATED WORK

In this section, we explain studies related to HACKSAW.

System debloating. Numerous researchers propose mechanisms to debloat systems software such as operating system kernels, unikernels, containers, and hypervisors. Kernel tailoring [47] reduces the kernel's attack surface by tuning it for a specific workload. In a development environment, it runs a target workload while recording kernel-level execution traces to figure out required kernel build configuration options and rebuilds the kernel according to them. Alharthi et al. [2] also confirm that such configuration-based kernel debloating is effective to nullify many kernel vulnerabilities. Cozart [45] improves configuration- and trace-based kernel debloating with instruction-level tracing from the early boot phase. Instead of figuring out a build configuration to rebuild the kernel, which is ad-hoc and often incomplete [28, 60, 80] and coarse-grained, FACE-CHANGE [36] and KASR [88] use the hypervisor to provide different kernel views (i.e., a restricted set of accessible or executable kernel memory pages) for each application based on its kernel-level execution trace. Also, temporal specialization [29], SHARD [1], and C2C [30] modify the kernel to provide a custom set of system calls to each application. In addition, other researchers rely on execution traces to debloat unikernels [46, 53, 61], container images [21, 69, 81], and device drivers [37, 40, 87].

Trace-based debloating mechanisms are highly effective in reducing the attack surface. However, they have a fundamental limitation because they test or sample workloads to generate execution traces. Thus, they cannot completely cover all possible cases, suffering from false removals. Unlike those approaches, HACKSAW is guided by the ground truth (i.e., whether certain hardware components exist), so it is accurate. Trace-based debloating and HACKSAW are orthogonal and can complement each other. In addition, like HACKSAW, delusional boot [59] relies on hardware information to debloat the hypervisor. Also, Cocoon [38] uses hardware information to debloat the kernel, but it rebuilds the kernel for every boot.

Driver isolation and bug finding. Device drivers are prone to have many security vulnerabilities [15, 25, 31]. However, we cannot simply remove them because they are critical to system operations [42]. To overcome this security problem, researchers propose driver isolation to ensure even if a driver is compromised, it cannot tamper with the main kernel. Various mechanisms have been used to realize driver isolation including Software Fault Isolation (SFI) [13, 55], user-mode drivers [8], isolated virtual machines [3, 41, 57, 58], and hardware-assisted intra-kernel isolation [56].

Researchers also try to automatically find bugs from device drivers to deal with this security problem. Some of them statically analyze device drivers to find bugs [7, 52]. Others fuzz the interfaces between applications and device drivers (e.g., `ioctl()`) [18, 44, 67, 89], and between hardware components and device drivers (e.g., MMIO, DMA) [39, 77–79, 90]. They further focus on fuzzing specific hardware: e.g., Bluetooth [26, 27, 73], USB [43, 63], and virtual device [9, 76]. Both driver isolation and bug finding are orthogonal to HACKSAW and can be adopted together to improve kernel security.

9 CONCLUSION

HACKSAW reduces the attack surface of the operating system kernel by specializing it for a target machine according to the hardware components attached to it. HACKSAW accurately removes inoperative drivers, modules, and kernel functions based on the hardware

device inventory and the dependency analysis without false removals. Our evaluation with seven different hardware platforms and 12 different source system images show that Hacksaw effectively reduces their attack surfaces while ensuring the validity and compatibility.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual.

[2] Mansour Alharthi, Hong Hu, Hyungon Moon, and Taesoo Kim. 2018. On the Effectiveness of Kernel Debloating via Compile-time Configuration. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[3] Sebastian Angel, Riad S. Wahby, Max Howald, Joshua B. Leners, Michael Spilo, Zhen Sun, Andrew J. Blumberg, and Michael Walfish. 2016. Defending against malicious peripherals with Cinch. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.

[4] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. 2023. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA.

[5] Patroklos Argyroudis and Dimitris Glynos. 2011. Protecting the Core: Kernel Exploitation Mitigations. *Black Hat Europe* (2011).

[6] Arm. 2021. Software Just Works on Arm-Based Devices.

[7] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual.

[8] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[9] Alexander Bulekov, Bandan Das, Stefan Hajnocz, and Manuel Egele. 2022. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA.

[10] Douglas Campbell and Chris Grevstad. 1985. A tutorial for make. In *Proceedings of the 1985 ACM Annual Conference on The Range of Computing*.

[11] Javier Martinez Canillas. 2012. Kbuild: the Linux Kernel Build System. *Linux Journal* (2012).

[12] Capstone. 2023. Capstone: The Ultimate Disassembly. https://www.capstone-engine.org.

[13] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT.

[14] Giacomo Catenazzi. 2023. LKDDb: Linux Kernel Driver DataBase. https://cateee.net/lkddb/.

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*.

[16] Jonathan Corbet. 2017. Restricting automatic kernel-module loading. https://lwn.net/Articles/740455/.

[17] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers*. "O'Reilly Media, Inc.".

[18] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[20] devicetree.org. 2021. Devicetree Specification Release v0.4-rc1. https://www.devicetree.org/specifications/.

[21] DockerSlim. 2022. DockerSlim. https://dockersl.im.

[22] FIRST. 2019. Common Vulnerability Scoring System v3.1: Specification Document. https://www.first.org/cvss/v3.1/specification-document.

[23] Foundeo Inc. 2023. Linux Kernel - Security Vulnerabilities in 2023. https://stack.watch/product/linux/linux-kernel/.

[24] Foundeo Inc. 2023. Microsoft Windows 10 - Security Vulnerabilities in 2023. https://stack.watch/product/microsoft/windows-10/.

[25] Archana Ganapathi, Viji Ganapathi, and David A Patterson. 2006. Windows XP Kernel Crash Analysis. In *Proceedings of the 20th Large Installation System Administration Conference (LISA)*.

[26] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2022. BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA.

[27] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2020. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*.

[28] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of 2017 11th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[29] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[30] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA.

[31] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT.

[32] Google Cloud. 2019. Migration to Google Cloud: Getting started. https://cloud.google.com/architecture/migration-to-gcp-getting-started.

[33] Jay Gordon. 2021. On Prem To The Cloud: Lift and Shift (Ep 2). https://devblogs.microsoft.com/devops/on-prem-to-the-cloud-lift-and-shift-ep-2/.

[34] Stephen J. Gowdy. 2023. The USB ID Repository. http://www.linux-usb.org/usb-ids.html.

[35] GRIMM. 2021. New Old Bugs in the Linux Kernel. https://blog.grimm-co.com/2021/03/new-old-bugs-in-linux-kernel.html.

[36] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. Face-Change: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[37] Zhongshu Gu, William N. Sumner, Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. DRIP: A Framework for Purifying Trojaned Kernel Drivers. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[38] Bernhard Heinloth, Marco Ammon, Dustin T Nguyen, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. 2019. Cocoon: Custom-Fitted Kernel Compiled on Demand. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS)*.

[39] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[40] Zhenghao Hu and Brendan Dolan-Gavitt. 2022. IRQDebloat: Reducing Driver Attack Surface in Embedded Devices. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[41] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.

[42] Asim Kadav and Michael M Swift. 2012. Understanding Modern Device Drivers. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK.

[43] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[44] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. Santa Clara, CA.

[45] 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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* 4, 1 (2020), 1–27.

[46] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Crete, Greece.

[47] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time

Kernel Tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[48] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.
[49] Linux Kernel CVEs. 2023. Linux Kernel Vulnerability Tracker. https://linuxkernelcves.com.
[50] Linux Test Project. 2023. LTP - Linux Test Project. https://linux-test-project.github.io.
[51] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.
[52] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR.CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, Canada.
[53] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than you Container. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China.
[54] Mohamad Mansouri, Jun Xu, and Georgios Portokalidis. 2023. Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. Melbourne, Victoria, Australia.
[55] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal.
[56] Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKCs. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[57] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA.
[58] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. Lausanne, Switzerland.
[59] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. 2012. Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*. Bern, Switzerland.
[60] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[61] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. Providence, RI.
[62] PCI SIG. 2019. PCI Express Base Specification Revision 5.0 Version 1.0. https://members.pcisig.com/wg/PCI-SIG/document/13005.
[63] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.
[64] Phoronix Media. 2023. Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking. https://www.phoronix-test-suite.com.
[65] Albert Pool and Martin Mares. 2023. The PCI ID Repository. https://pci-ids.ucw.cz.
[66] Alexander Popov. 2021. Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel. https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.
[67] Ivan Pustogarov, Qian Wu, and David Lie. 2020. Ex-vivo dynamic analysis framework for Android device drivers. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[68] Srikanth Rangavajhala and Prasanna Raghavendran. 2021. Lift and shift: Rehost your workload on AWS to accelerate your cloud journey. https://docs.aws.amazon.com/prescriptive-guidance/latest/strategy-rehosting/welcome.html.
[69] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany.
[70] Tristan Ravitch. 2023. travitch/whole-program-llvm: A wrapper script to build whole-program LLVM bitcode files. https://github.com/travitch/whole-program-llvm.
[71] Petter Reinholdtsen. 2013. Modalias strings - a practical way to map "stuff" to hardware. http://people.skolelinux.org/pere/blog/Modalias_strings___a_practical_way_to_map__stuff__to_hardware.html.
[72] Rami Rosen. 2013. Resource management: Linux kernel Namespaces and cgroups. http://www.haifux.org/lectures/299/netLec7.pdf.
[73] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.
[74] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.
[75] Michael S. 2019. Linux kernel module autoloading. https://duasynt.com/blog/linux-kernel-module-autoloading.
[76] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual.
[77] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. 2022. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA.
[78] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[79] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.
[80] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the 6th Conference on Computer Systems (EuroSys)*.
[81] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. CNTR: Lightweight OS Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.
[82] The kernel development community. 2023. Development tools for the kernel. https://www.kernel.org/doc/html/latest/dev-tools/index.html.
[83] The kernel development community. 2023. Device drivers infrastructure. https://www.kernel.org/doc/html/latest/driver-api/infrastructure.html.
[84] The kernel development community. 2023. Platform Devices and Drivers. https://www.kernel.org/doc/html/latest/driver-api/driver-model/platform.html.
[85] The kernel development community. 2023. Seccomp BPF (Secure Computing with Filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.
[86] UEFI Form, Inc. 2022. ACPI Specification 6.5. https://uefi.org/specs/ACPI/6.5/.
[87] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. 2021. LightBlue: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual.
[88] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.
[89] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA.
[90] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. 2022. Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.