KSplit: Automating Device Driver Isolation

Yongzhe Huang[*1], Vikram Narayanan[*2], David Detweiler[2], Kaiming Huang[1], Gang Tan[1], Trent Jaeger[1], and Anton Burtsev[2,3]

[1]Pennsylvania State University
[2]University of California, Irvine
[3]University of Utah

Abstract

Researchers have shown that recent CPU extensions support practical, low-overhead driver isolation to protect kernels from defects and vulnerabilities in device drivers. With performance no longer being the main roadblock, the complexity of isolating device drivers has become the main challenge. Device drivers and kernel extensions are developed in a shared memory environment in which the state shared between the kernel and the driver is mixed in a complex hierarchy of data structures, making it difficult for programmers to ensure that the shared state is synchronized correctly. In this paper, we present KSplit, a new framework for isolating unmodified device drivers in a modern, full-featured kernel. KSplit performs automated analyses on the unmodified source code of the kernel and the driver to: 1) identify the state shared between the kernel and driver and 2) to compute the synchronization requirements for just this shared state for efficient isolation. While some kernel idioms present ambiguities that cannot be resolved automatically at present, KSplit classifies most ambiguous pointers and identifies ones requiring manual intervention. We evaluate our solution on nine subsystems in the Linux kernel by applying KSplit to 354 device drivers and validating isolation for 10 drivers. For example, for a complex Ixgbe driver KSplit requires only 53 lines of manual changes to 2,476 lines of automatically generated interface specifications and 19 lines of changes to the driver's code. The KSplit analysis of the 354 drivers shows a similar fraction of manual work is expected, showing that KSplit is a practical tool for automating key tasks to enable driver isolation.

1 Introduction

Device drivers have long been and continue to be a major source of defects and vulnerabilities in modern kernels [19, 32, 50, 65]. Drivers are expected to support a variety of complex protocols and comply with numerous kernel conventions [23, 76, 77], creating challenges in ensuring that device drivers operate correctly in the face of concurrent and asynchronous accesses on multiple CPU cores. In addition, while the core kernel is relatively stable, the number of kernel extensions and device drivers is large and continues to grow. A modern Linux 5.12 kernel contains around 8,960 device drivers that account for 67.7% of the kernel source [3], a number that has nearly doubled since 2013. With a rate of 80,000 commits a year, defects and vulnerabilities are an inherent part of the fast growing and evolving driver codebase.

The recent availability of hardware features for efficient isolation [1, 5] and system support that leverages such features [40, 43, 47, 61, 63, 82] have made low-overhead device isolation frameworks practical [66, 68]. The upcoming hardware extensions, e.g., native page-granularity support for isolation of kernel code [5], and 16 byte-granularity isolation with memory tagging extensions (MTE) [1], which are key for enabling low-overhead SFI implementations [53], will reduce overheads of isolation even more.

Despite availability of low-overhead isolation mechanisms the task of isolating existing driver code remains challenging. For decades, device drivers and kernel extensions have been developed in a shared memory environment of a monolithic kernel, where they freely exchange references to large and complex data structures (e.g., many data and pointer fields, hierarchical, and cyclic) that mix the state of the driver and the kernel. Isolating a driver requires a careful analysis of the flow of execution between isolated subsystems to identify how the complex state of the system is accessed on both sides of the isolation boundary.

Recent techniques to isolate legacy driver code utilize manual analysis of complex kernel-driver dependencies [18, 62, 66, 68, 80], requiring an immense decomposition effort that limits their applicability, and proposed techniques to automate such analyses [33, 72] only address a small fraction of the task. For example, LXFI, an SFI framework, utilized an iterative procedure to identify all the state required for execution of a driver, iteratively annotating the missing parts of the shared state [62]. The scale and complexity of modern drivers make such manual analysis impractical. In the Linux kernel, even simple drivers like MSR that provide an interface to model specific CPU registers (MSRs) require analysis of 459 functions and around 10,000 object fields that are transitively reachable from the 21 functions of the driver interface. A more complex network driver, Ixgbe, requires analysis of

[*]Contributed equally
[†]Work done partly at the University of California, Irvine

5,782 functions and over 900,000 object fields—a number that is well beyond the reach of manual analysis. Decaf [72] and Microdrivers [33] took initial steps towards automated analysis for driver isolation prior to the advent of efficient isolation hardware, so these works focused on techniques to isolate only non-critical path driver functionalty. As a result, many challenging parts of the drivers, e.g., interrupt handlers and optimized data plane functions, remain inside the unisolated kernel. In addition, these techniques do not aim to minimize data synchronization effort and leave several manual tasks.

In this paper, we present KSplit, a new framework for isolating device drivers in the Linux kernel. KSplit performs a collection of static analyses on the source code of the kernel and the driver to generate the synchronization code that is required to execute the driver in isolation. Specifically, KSplit identifies the shared state that is accessed by both driver and kernel, computing how this state is accessed on either side of the isolation boundary, and how it should be synchronized on each kernel-driver invocation, or when a shared synchronization primitive (e.g. a spinlock or an RCU) is invoked. The result of the analysis is a collection of procedure call specifications in the KSplit interface definition language (IDL). The KSplit IDL compiler then generates glue code that ensures synchronization of data structures between isolated subsystems. Some kernel idioms, such as concurrency and complex data structures, present ambiguities that cannot be resolved automatically at present, so KSplit also identifies these specific problems for developers to focus their effort. This allows one to take an existing driver and produce the data synchronization code necessary to run the driver in isolation, automatically if possible, and identify remaining tasks that require manual intervention, if needed.

Kernel software presents several challenges for developing accurate and scalable analyses for automating the isolation of legacy drivers that we address in the design of KSplit[1].

First, modern kernels have evolved to share fine-grained access to large, hierarchical data structures with their drivers, which enables joint, optimized operation over shared state using complex memory references. To compute shared state accurately, KSplit employs a field-sensitive data flow analysis using a modular alias analysis to identify shared fields while accounting for memory references accurately. To compute shared state scalably, KSplit provides a two-stage analysis to identify the kernel functions that could possibly share access to a data structure with a particular driver, enabling the accurate analysis to be targeted to the relevant subset of the kernel.

Second, modern drivers provide concurrent access to a variety of functionality, even using kernel code, which complicates the challenge of ensuring that the shared state is synchronized correctly when the driver is isolated. KSplit provides algorithms to ensure correct synchronization of shared state for driver invocations, nested calls to kernel functions by drivers, and a variety of concurrency primitives, including spin and sequential locks, read-copy-update (RCU), mutexes, and atomics. KSplit provides an analysis to identify concurrency primitives that operate over shared data, finding that such primitives rarely cross the kernel-driver boundary.

Third, kernels utilize a wide range of low-level idioms that create ambiguities for marshaling in synchronization, e.g., sentinel and sized arrays, tagged and anonymous unions, self-referential data structures like linked lists, etc. To separate complete drivers, many of these ambiguities need to be resolved automatically. KSplit partitions these data structures into classes to apply algorithms to determine whether marshaling requirements can be inferred or not. KSplit is able to automate most cases and provide warnings for the rest.

We develop KSplit for the Linux kernel and a recent device driver isolation framework, LVDs [68]. KSplit is a fully parallel analysis that takes only a few seconds to complete for simple drivers, and completes within minutes for complex device drivers like `Ixgbe`. We evaluate driver isolation using KSplit on 10 Linux device drivers, intentionally choosing device drivers representing a wide variety of functionality and kernel programming idioms. Simple device drivers like MSR can be isolated with no changes to their code, and only 2 lines of IDL changes are required to resolve ambiguities in the driver's IDL. More complex drivers like Ixgbe require less than 20 lines of driver code changes and only 53 lines of IDL changes for the 2,476 lines of device interface definitions. We also apply the KSplit analysis to 354 drivers, finding that the amount of manual effort is expected to be a similar fraction of the driver size. Drivers isolated using KSplit leverage the low-overhead hardware and software isolation mechanisms, remaining with 5.4–18.7% of non-isolated systems' performance. Our experience with isolating device drivers confirms that KSplit is a practical tool for enabling isolation of complete, legacy device drivers through the use of emerging low-overhead hardware and software isolation mechanisms.

2 Background: Device Driver Isolation

Over the years a range of techniques to isolate kernel extensions explored execution of device drivers in clean-slate microkernels [10, 12, 13, 27, 30, 35, 39, 44–46, 49, 57] and virtual machines [14, 15, 31, 34, 55, 70, 75], re-implementing device drivers in safe programming languages [9, 11, 37, 48, 56, 67, 84], developing backward-compatible driver execution frameworks [8, 17, 22, 24, 29, 36, 38, 42, 52, 71, 81, 83, 86], and finally isolating unmodified driver code with hardware [33, 66, 68, 80] and software [18, 25, 62] mechanisms. While it is possible to enforce isolation of the driver code through programming language safety [11, 48, 56] and formal verification [7, 20], to achieve isolation of unmodified drivers, driver isolation frameworks rely on either hardware isolation mechanisms (e.g., segmentation, paging, EPT switching,

[1] KSplit is developed for Linux, but our techniques can be applied to other commodity kernels.

core-isolation [66], or techniques of software fault isolation (SFI) [18, 25, 62, 85]. instrumentation of control flow and memory instructions [18, 25, 62].

The main challenge in isolating legacy drivers is to provide controlled access to the state that is shared by the kernel and the isolated driver. Commodity operating systems allow kernels to share an address space and hence its entire state with drivers, implementing driver operations on objects jointly accessible to both the kernel and the driver. Often these objects have a complex, hierarchical structure, e.g., `sk_buff` network packets, but only a fraction of these objects (i.e., a small subset of their fields) are accessed by both the kernel and the driver in practice, forming the shared state. In order for the isolated driver to work correctly, KSplit must identify this shared state comprehensively, but to provide efficient isolation, KSplit must not overapproximate the shared state significantly.

Hardware and SFI frameworks take different approaches to protecting access to the state shared between the kernel and the driver. Hardware approaches control access by executing the driver on an isolated copy of the shared state that is synchronized with the kernel on each driver invocation [33, 66, 68, 80]. SFI approaches, in contrast, execute the driver and the kernel on a single copy of the shared state. This eliminates the need for maintaining two copies of the shared state, but requires access-control checks on each memory access to the shared state [62]. To provide fine-grained access control on the kernel state, SFI systems implement a concept of "capability tables" that allow quick byte-granularity lookup of each kernel field accessible to the driver [62].

Irrespective of the isolation mechanism, however, both solutions require analysis of which state can be accessed by the driver and the kernel and when each access is allowed [62]. Decaf [72] and Microdrivers [33] took a first step in automated generation of shared state for isolated drivers by computing the state accessed by the driver on each invocation. However, not all of this state is shared with the kernel, as we find that drivers operate on a significant amount of state that is private to the driver. In addition, these projects only decomposed the non-critical path driver code into isolated domains to retain reasonable performance.

Historically, isolation in the kernel remained prohibitive due to the high overhead of hardware and software isolation mechanisms. Recent CPUs, however, signal the growing support for low-overhead isolation primitives. Extended page-table (EPT) switching with VM functions [6] and user-space memory protection keys (MPKs) [6] already provide support for memory isolation with overheads comparable to system calls for Intel machines [43, 63, 68, 82], and the next generation of Intel machines promises to extend MPK with native support for isolation of ring 0 code [5]. Similarly, the newest ARM CPUs provide support for 16 byte-granularity isolation with memory tagging extensions (MTE) [1], which is key for enabling low-overhead SFI implementations [53].

For device driver isolation, recent work has shown that using domain-based isolation can be practical. LXDs [66] and LVDs [68] develop a Nooks-like isolation framework using extended page tables (EPTs) to improve boundary crossing performance, providing an interface definition language (IDL) for specifying which data requires synchronization from driver interfaces. This work demonstrates the potential for efficient hardware-based protection domain isolation of legacy drivers. However, such isolation required a significant manual effort to configure IDL definitions for complete drivers. While previous work [33, 72] proposed a method to generate the base IDL, configuring the marshalling requirements for a variety of complex data types and handling concurrency was performed manually. While SFI does not require synchronization on boundary crossings, SFI methods must compute essentially the same information to enable correct isolation for good performance.

A variety of projects have explored techniques to automate various aspects of decomposing user-space programs [16, 21, 41, 58–61, 74, 87–89], called *privilege separation* [78], but these techniques fail to address issues critical to isolating kernel code. For example, PtrSplit [59] proposed techniques to compute marshalling requirements for objects based on runtime tracking, but this adds non-trivial overhead. In addition, these techniques are not designed to handle multi-threaded programs like a kernel.

3 KSplit Overview

KSplit transforms complete, shared memory device drivers into equivalent drivers that can execute in an isolated domain and on an isolated copy of the driver state. Specifically, KSplit identifies the subset of the kernel state that is required for an isolated driver to run and derives how this state has to be synchronized on invocations between the kernel and the driver that cross the isolation boundary and at the points where the driver uses concurrency primitives[2], e.g., atomics, spinlocks, mutexes, ready-copy-update (RCU), etc.

For example, when the kernel submits a network packet to a network device with the `ndo_start_xmit()` function, KSplit ensures that all the shared fields (i.e., between the kernel and driver) of all the data structures that are recursively reachable from the two input arguments (i.e., `sk_buff` and `net_device`) and all global kernel variables are synchronized with the driver. After the invocation completes, the fields updated by the driver are synchronized back to the kernel. Nested invocations into the kernel also trigger synchronization to ensure that the kernel and driver use the current state. If the driver code uses a concurrency primitive that is shared with the kernel, e.g., a global lock, like the `rtnl_lock` used by network device drivers to register with the kernel, KSplit synchronizes the state of the driver with the kernel on entry and exit from the atomic region to maintain current copies in both domains.

[2]To distinguish between the synchronization of shared state in general with primitives to synchronize state used in concurrent operations, we refer the latter as *concurrency primitives* in this paper.

KSplit provides software analysis algorithms to isolate legacy drivers to achieve these goals that 1) compute the subset of the kernel state that is accessed by the driver (i.e., the shared state) and 2) synchronize that shared state on cross-domain invocations and concurrency primitives that access shared state. While appearing to be conceptually straightforward, isolating legacy drivers is complicated by several factors caused by how drivers are currently deployed in monolithic kernels, specifically:

Complex shared state Kernel data structures often consist of a large number of fields and may be referenced in a variety of ways. The `sk_buff` structure that represents a network packet has 66 fields (5 pointers and 2 offsets) through which 3,132 fields (1,214 pointers) are recursively reachable in other data structures. The kernel and driver operate jointly on only a small fraction (52 shared fields) of these fields. In addition, like many kernel data structures, the `sk_buff` structure is accessed through complex memory references. For example, some `sk_buff` pointers are used to ensure in-place access to parts of the network packet, i.e., `head` and `data` mark the beginning of the packet header and data regions from which the packet is assembled, respectively.

To compute shared state accurately under these requiements, KSplit employs a field-sensitive data flow analysis using a modular alias analysis to capture field references common between the kernel and driver. To do this efficiently, we apply the parameter tree approach [59], which computes aliases intra-procedurally [79] and propagates those alias results inter-procedurally. This approach was employed previously in user-space privilege separation [59]. However, user-space privilege separation aims to isolate sensitive data selected manually by programmers, whereas KSplit needs to identify the data shared between the kernel and a driver automatically. Prior techniques to estimate sharing between the kernel and a driver [33, 72] greatly overestimate shared data because they collect all the fields that the driver will access instead of those that are actually shared.

Size and complexity of the kernel In order for the isolated driver and kernel to operate correctly, we must identify all the shared state. Using a sound alias analysis, we can over-approximate the shared state, but the kernel is too large (e.g., contains 53,000 functions) to apply the field-sensitive analysis needed to compute shared state accurately. KSplit handles this challenge by first performing an analysis to identify the subset of kernel functions that can access the state involved in interaction with the driver. Then, KSplit performs an accurate shared state analysis on this subset of the kernel functions along with the driver.

Concurrency and parallelism KSplit must ensure that the kernel and the isolated driver operate on current shared state regardless of how the kernel and driver interact. The kernel, however, invokes functions of the driver in parallel on multiple CPUs and device drivers are concurrent and fully-reentrant. As a result, it is possible that the driver updates the shared state that is concurrently accessed by the kernel or vice versa, using one of various concurrency primitives. For example, most drivers use the read-copy-update (RCU) synchronization pattern to synchronize their state across multiple invocations in a lightweight manner, e.g., the Ixgbe network driver holds an `rcu_read_lock` to access the ring statistics to prevent deallocation of driver queues by a concurrent thread. However, many drivers rely on atomic primitives and critical sections (e.g., Ixgbe communicates state updates to the New-API (NAPI) state to the softirq framework with atomic variables). Finally, some device subsystems rely on global locks (e.g., `rtnl_lock` in the network subsystem) during the driver registration.

KSplit leverages a critical observation that synchronization mechanisms rarely cross the driver-kernel boundary, e.g., out of 73 uses of concurrency primitives in the Ixgbe driver only 3 atomic primitives synchronize state across the isolation boundary. We develop a collection of algorithms that carefully classify shared and private critical sections for a range of kernel concurrency primitives (mutexes, spinlocks, sequential locks, atomic primitives, and RCU locks). For shared concurrency primitives, KSplit computes the state that is accessed within the critical section and requires synchronization.

Low-level C idioms The kernel code utilizes a range of low-level idioms that create ambiguities for static analysis (Figure 1). For example, device drivers rely on sentinel values (e.g., `null`) to represent variable size arrays, e.g., the PCI subsystem uses the `pci_id_table` array to store a set of devices supported by a particular driver (Figure 1a). Further, the lack of a fast array or vector abstraction forces the kernel to use references in place of arrays and keep the length as a separate field. Some memory regions like user and device I/O memory requires special treatment when passed into an isolated driver (Figure 1c). Tagged and anonymous unions are used by the driver to implement polymorphic functions that can take generic arguments of a union type (Figure 1e). KSplit provides support for these cases and the necessary IDL annotations and library support to generate correct code.

Prior approaches assumed that programmers would provide annotations to resolve ambiguities in marshaling manually for most cases [33, 51, 62, 66, 68], but that is impractical when isolating complete device drivers. Instead, KSplit takes the opposite approach, aiming to resolve ambiguities in most cases and providing warnings in the remaining cases. For example, `char *` references, as for the `head*` and `tail*` fields in the `sk_buff` data structure, may refer to singletons, arrays, strings, or even other data types (e.g., for type casts). KSplit utilizes a series of classification methods to distinguish among these cases automatically, enabling nearly all ambiguities to be resolved for the drivers we have isolated.

Prior work Microdrivers [33], Decaf [72], and FGFT [51] developed static analysis aimed at isolation of legacy driver code. Due to the sheer complexity of the whole driver analysis, these past approaches were limited to isolating only select driver functions (e.g., non-critical path), and supported only

```c
struct pci_dev { // sized array
  struct resource resource[DEVICE_COUNT_RESOURCE];
};

static const struct pci_device_id ixgbe_pci_tbl[]
  = {
    { PCI_VDEVICE(INTEL, IXGBE_DEV_ID_82598),
      board_82598 },
    { }, /* sentinel */
  };
```

(a) Sized and sentinel arrays

```c
#define skb_shinfo(SKB) \
    ((struct skb_shared_info *)(SKB->end))

static inline void
   *blk_mq_rq_to_pdu(struct request *rq)
{
  return rq + 1;
}
```

(b) Collocated data structures

```c
ssize_t msr_read(struct file *file,
                  char __user *buf, ...)

dev->bar = ioremap(pci_resource_start(pdev, 0),
                 8192);
```

(c) Special memory regions.

```c
struct skb_shared_info {
  struct sk_buff *frag_list;
};
```

(d) Recursive data structures

```c
union acpi_object {
  acpi_object_type type; /* tag */
  struct {
    acpi_object_type type;
    u64 value;
  } integer;
  ...
};
```

(e) Tagged unions

```c
static int ixgbe_set_mac(struct net_device *netdev,
                         void *p) {
  struct sockaddr *addr = p;
  memcpy(netdev->dev_addr, addr->sa_data,
       netdev->addr_len);
  ...
}
```

(f) Opaque pointers

Figure 1: Code idioms typical of the Linux kernel

a limited subset of kernel idioms. KSplit leverages advances in static analysis, specifically, a combination of an accurate Program Dependence Graph representation and modular alias analysis with parameter trees [59]. This allows KSplit to scale the analysis and implement isolation of the entire driver. A clean separation of shared and private state allows us to scale static analysis and resolve almost all ambiguous annotations required for marshalling of the low-level driver code.

3.1 Threat Model and Security Goal

The goal of KSplit is the same as the majority of prior research on driver isolation [35, 66, 68, 80] in that KSplit aims to improve kernel reliability, i.e., prevent flaws in the driver domain, such as memory errors, from affecting the rest of the kernel. We trust that the kernel domain is free of software flaws, but assume that the driver domain may contain flaws that, for example, may result in writes to kernel memory possibly causing the kernel to crash.

We leave the feasibility analysis of whether KSplit driver isolation prevents attacks originating from a driver as future work. We note that LXFI [62] prevents certain driver-originated attacks by generating dynamic checks based on user-specified safety conditions at the kernel-driver boundary. However, identifying and specifying safety conditions for individual drivers are labor-intensive tasks and a range of security attacks are still possible, such as resource exhaustion (e.g., driver can allocate objects to consume memory), protocol violations (e.g., driver can unregister itself from the kernel), and even use-after-free (e.g., driver can trigger deallocations of objects reachable from the kernel in an unexpected way). We, however, believe that KSplit is a critical step towards shaping the foundation of the future isolation mechanisms. We plan to study what security guarantees may be possible to achieve automatically as future work, e.g., by extending our analyses to produce changes to the kernel-driver boundary required to address the security problems above.

Finally, speculative execution and side-channel attacks are out of the scope of this work as well.

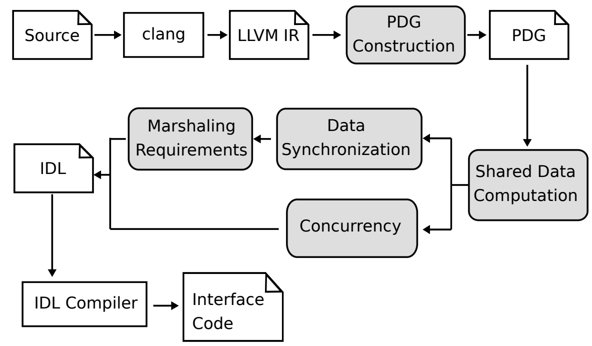

Figure 2: KSplit workflow.

4 KSplit Static Analysis

Figure 2 presents KSplit's workflow. KSplit takes the source code (i.e., the code of the kernel and a device driver) as input and converts it into LLVM IR using Clang, LLVM's front-end. KSplit then provides analyses to: (1) *identify shared data* between the kernel and the driver; (2) *compute data synchronization* on each boundary crossing for that shared data; (3) compute data synchronization for *concurrency primitives* that access shared data; and (4) *infer marshaling requirements* for data types where such requirements are ambiguous, e.g., tagged unions, void pointers, arrays, linked data structures, etc. The result of the analysis is a collection of definitions for the KSplit interface definition language (IDL) compiler. For some cases whose IDL configuration (e.g., size and/or format) remains ambiguous after analysis, KSplit generates warnings for developers to resolve the ambiguity. These warnings must be resolved by developers to obtain a working IDL. The IDL compiler then generates glue code that ensures synchronization of data structures between isolated subsystems.

In this section, we present KSplit's core static-analysis algorithms to address the aforementioned problems. The algorithms are designed to solve these problems in general, but the C language is ambiguous about some key information required by the algorithms (e.g., pointer type information). We defer to Section 5 for a discussion of how we leverage C programming idioms used in the Linux kernel to resolve these ambiguities in many cases. While some of these idioms are commonly applied in C programs, some idioms may need to be replaced for other kernels.

4.1 Program Dependence Graph

KSplit reasons about the kernel and drivers using an interprocedural program-dependence graph (PDG) [59]. PDG represents individual LLVM instructions as nodes with edges that capture control and data dependence among instructions. An instruction n_1 is *control dependent* on n_2 if, intuitively, n_2's outcome decides whether n_1 gets executed [26]. An instruction n_1 is *data dependent* on instruction n_2 if n_1 uses some data produced by n_2. Data dependence is critical for determining how the data structures that are exchanged between the driver and the kernel are used in cross-domain invocations. Specifically, KSplit computes how the objects are used by each side of the isolation boundary to compute data synchronization requirements as described in Section 4.3. In particular, we need to find all operations that may read or write data, which should be marshaled across a boundary.

Scaling alias analysis with parameter trees A common type of data dependence happens when an instruction writes to a piece of memory from which another instruction reads. Computing such memory-related data dependence requires *alias analysis*, which computes the variables or expressions that may reference (i.e., point to) the same memory object, which are called *aliases*. We need to compute aliases in KSplit because we need to detect all objects that may be accessed by both the kernel and drivers. Further, the isolation of the driver code further requires an interprocedural alias analysis as both the kernel and driver code may pass pointers to data objects across function boundaries as well as through global variables. The alias analysis problem is known to be undecidable; devising a precise analysis that is guaranteed to capture all aliases and scales well is a challenge. Current interprocedural alias analysis techniques (e.g., [54, 79]), however, do not scale to low-level kernel code with its complex uses of memory references. Instead, we propose to deploy a modular form of alias analysis that enables us to manage scalability more effectively.

In the KSplit approach to modular alias analysis, we employ SVF [79] to compute aliases intra-procedurally and propagate those alias results inter-procedurally using the parameter tree approach [59], allowing us to efficiently compute memory dependence across function boundaries in a context-insensitive way. Specifically, it first constructs PDGs for each function of the program (which includes intra-procedural memory dependence) and then glues them together by connecting actual parameter trees for arguments at function call sites and formal parameter trees for parameters; details can be found in [59].

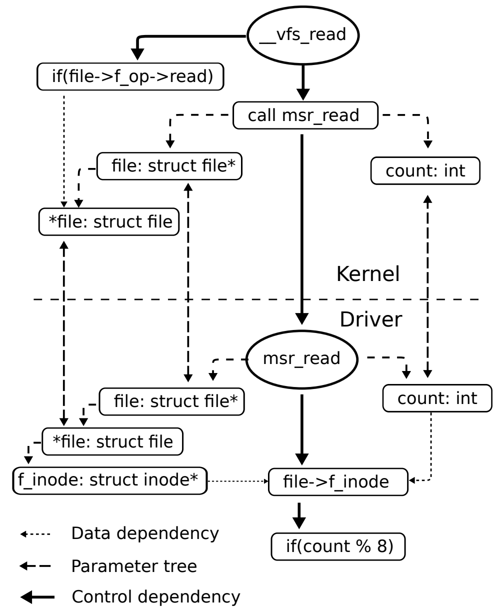

Figure 3: Partial PDG for the `msr_read()` function which is invoked with the `call` instruction from the `__vfs_read()` function.

To illustrate the idea of parameter trees, consider the `msr_read()` function of the MSR driver. For each argument of the function we construct a parameter tree that represents storage locations that the callee can access. For example, Figure 3 shows a parameter tree for two arguments of the `msr_read()` function: 1) the `file` argument of type `struct file *` and 2) the `int` argument `count`. The parameter tree for the `file` argument has a root node labeled "`file:struct file*`" for representing the storage of the pointer, and a child node labeled "`*file:struct file`" for the memory region that the pointer points to. The references of each storage location in the program are connected with corresponding tree node through data dependency edges. We note that for brevity the figure does not show the fields of the file struct; the actual representation represents each field by a separate node for field sensitivity.

4.2 Computing Shared and Private Data

Accurate separation of shared and private state is critical for precision and scalability of KSplit analyses. However, the size of the kernel makes it impractical to perform an accurate analysis to find the shared state at the level of fields (i.e., field-sensitive analysis). On the other hand, the kernel's use of interrupt handlers makes it difficult to ensure that all the code that may impact a particular driver interface invocation has been accounted for. For example, an interrupt handler does not have an explicit caller and is thus unreachable in a typical control-flow graph (CFG) from the driver or regular kernel code. It only runs in response to the corresponding interrupt.

As a result, we develop a shared state algorithm that first determines the scope of code in the kernel and driver to consider (i.e., the functions and data types that may be shared), as described in steps (1) and (2) of the detailed algorithm below. Then, we perform an accurate, field-sensitive analysis on the PDG leveraging the modular alias analysis above to capture the shared state of the kernel and driver in terms of data structure fields.

The detailed algorithm steps are as follows: (1) the algorithm computes a set of struct types that are accessible by both sides of the isolation boundary. This is performed by collecting all the struct types that are accessible transitively through interface function parameters, global variables, and interrupt handlers. These struct types are referred as *shared struct types*. (2) For each shared struct type, we identify the functions in the kernel and driver that contain variables whose type matches one of the shared struct types. The functions accessible from isolation boundary in step (1) and those found in step (2) are used to compute the shared state in (3) below. Steps (1) and (2) do not use the CFG and work even for interrupt handlers (unreachable in the CFG). (3) For each set of variables that match a shared struct type, we use the PDG to analyze the accesses via the variables to collect the fields accesses for that type. (4) For each field, if the field has accesses from both the kernel and the driver, we consider the field as shared. Otherwise, the field is private.

The output of the algorithm is a set of shared struct types associated with their shared and private fields. For illustration, the struct `net_device` type contains the following fields (among others): `wanted_features`, `features`, and `hw_features`. By analyzing the `ixgbe` driver and the kernel code, our analysis determines that the `features` field has accesses from both the driver and the kernel, while the other two fields have only accesses in the kernel. Therefore, our algorithm decides that `features` is shared, while the other two are kernel private.

This algorithm relies on two assumptions. First, in step (2), we assume that any state shared between the kernel and driver are accessed using one of the shared struct types from step (1). While this is not guaranteed, the kernel generally obeys typing for the types shared with drivers. If we miss a data type, we may under-approximate shared state causing correctness issues, but we have not found any exceptions to date. Second, we rely on the observation that the type of a composite object correlates with how it is shared across the isolation boundary. In other words, it is uncommon for different instances of the same composite type to be either shared or private; e.g., if a device driver accesses the `inode` field of the `struct file *` object, it is likely that `inode` is shared for all instances of the `struct file *` type. Thus, the analysis cannot determine whether a field of one instance is shared while the same field of another instance is private. The algorithm may over-approximate the shared state, which may cause unnecessary data synchronization but does not affect correctness.

4.3 Cross-Domain Synchronization

When a function invocation crosses the domain boundary, KSplit synchronizes the shared state that is required by the callee domain to execute the call. Similarly, when the function returns, the changes the callee made to any shared state have to be synchronized back to the caller reflecting updates on its copy. We develop *parameter access analysis* that computes all data structures and their fields that require synchronization.

Basic parameter access analysis At a high level, this algorithm tracks the parameter reads that require data to be synchronized on calls and parameter writes that require data to be synchronized on responses for each cross-domain invocation and any functions reachable from that invocation. Algorithm 1 presents a worklist-based algorithm: 1) for each function in the worklist, it performs an intraprocedural parameter access analysis; 2) it collects call instructions in the function being analyzed and performs an interprocedural analysis; and 3) it repeats steps (1) and (2) until the analysis reaches a fixpoint (when the worklist becomes empty). Algorithm 1 computes field usage that is dependent on parameters passed between domains only. Dependence is computed using the parameter tree alias analysis to ensure an overapproximation.

Algorithm 1: Parameter access analysis

Input: G is a PDG, T is a parameter tree, f is the target function of a cross-domain call
Output: Access Information Map AM

initialize AM to be empty
$worklist \leftarrow \{f\}$
while *worklist is not empty* **do**
  $f_1 \leftarrow remove\_any(worklist)$
  for *node n in T* **do**
    for *instruction i in* f_1 **do**
      if *G has a dependence edge from i to n* **then**
        $AL \leftarrow$ the edge's access label
        $AM[n] \leftarrow AM[n] \cup AL$
      else if *i calls* f_2 **then**
        $worklist \leftarrow worklist \cup \{f_2\}$
      end
    end
  end
end

The analysis goal is to compute a set of *access labels (AL)* for each parameter tree node of a function parameter. The access label of a node represents how the storage represented by the node is used by the callee of a cross-domain call (*READ*/*WRITE*). We further define a global map *AM*, which maps from parameter tree nodes to sets of access labels *AL*. For example, if there is a read access to the `f_inode` field of the `file` data structure, we associate a *READ* label with the parameter tree node that represents the storage of the `f_inode`. After *AM* is computed, the fields for shared state corresponding to nodes with the *READ* label are copied from the caller to the callee when the call happens, and those for shared state with the *WRITE* label are copied from the callee to the caller when the callee returns.

The previous analysis generates correct state to synchronize, but might include unnecessary fields because of nested boundary crossings, which cause the call-graph transitive closure to include functions from both sides of the isolation boundary. KSplit distinguishes reads and writes of different domains and avoids sending shared data to a callee if the data is only used in the caller's domain due to a nested call. Similarly, KSplit avoids copying shared data back to the caller if the writes only occur in the caller domain, To do this, KSplit removes shared fields accessed only in the caller domain from the closure computed in Algorithm 1. For the above example, suppose the driver function d reads shared field `fd1` and k' reads shared field `fd2`. The previous analysis determines both `fd1` and `fd2` need to be sent when k calls d. However, our optimization distinguishes the two reads and sends only `fd1`.

4.4 Critical Sections and Atomic Primitives

Modern device drivers are often invoked in parallel on all CPUs of the system, and are fully concurrent outside of small critical sections. Kernel and drivers synchronize accesses to the shared state through a variety of concurrency mechanisms provided by the kernel: atomic operations, spinlocks, sequential and reader/writer locks, read-copy-update critical sections (RCU), etc. To support correct execution of an isolated driver, we provide support for concurrency primitives across the isolation boundary. We identify two large classes of concurrency primitives: locking and lock-free (i.e., atomic operations). For atomic update primitives, e.g., `atomic_inc()`, `atomic_set()`, we perform all updates on the primary copy of the data maintained in the kernel; i.e., drivers call outside to update the primary copy. For synchronization primitives that acquire and release a lock (we support spinlocks, seqlocks, RCU, reader/writer locks, and mutexes), we compute the state that is accessed in each critical section and synchronize it across the isolation boundary. To enforce atomicity across isolated domains, we rely on the mechanism similar to combolocks [33].

The high-level steps for the analysis are as follows: 1) identify *shared* critical sections where cross-subsystem synchronization is required; and 2) collect read/write accesses to shared data in critical sections.

Identifying critical sections To identify critical sections, we perform a search in the CFG of the program, looking for a set of function invocations that implement critical section synchronization primitives, e.g., `spin_lock`, `mutex_lock`, etc. For each call to a function marking the beginning of a critical section we follow the CFG to identify a matching invocation that marks its end, i.e., `spin_unlock` for `spin_lock`. Next, we use alias analysis to check whether the beginning (lock) and end (unlock) use the same lock. Finally, we output only critical sections defined by lock/unlock call pairs found by the CFG that are associated with the same lock.

Shared data accesses in critical sections Given a critical section, we identify all shared state that is modified within the critical section. Our goal is to: 1) classify critical sections and atomic operations as private or shared, i.e., whether the data accessed is private or shared, and then 2) if the critical section operates on shared data, compute the state required for correct synchronization. Specifically, we identify read and write accesses to shared data from inside the critical section (similar to Algorithm 1). For read accesses we ensure that tha state is synchronized right after entering the critical section—this ensures that the code inside operates on a consistent, fresh copy of the state. For write accesses we synchronize all updates by sending it to the other side of the isolation boundary right before exiting the critical section.

Handling optimized primitives KSplit has support for a variety of concurrency primitives that are optimized to reduce the use of locking. In most cases, such as sequential locks, the main issue is to determine the corresponding reader and writer critical sections accurately without explicitly locking. For example, we describe how KSplit handles RCU primitives. An RCU lock is often used in manipulating linked list data structures inside the kernel to enable multiple readers and a single writer to access the same data structure concurrently, which reduces the time-consuming lock obtain/release operations. In KSplit, we consider the non-preemptable reader implementation of RCU locks. In this implementation, the start and end of a reader section is defined by calls to `rcu_read_lock` and `rcu_read_unlock` functions, respectively. The reader critical section disables preemption. For an RCU writer, KSplit searches for the call sites of functions that may update the pointed data of an RCU pointer such as `rcu_assign_pointer` and `rcu_replace_pointer`. After identifying those reader and writer sections, the same synchronization algorithm as before is used. While this design impacts the optimization obtained by RCU locks, fortunately RCU locks are rarely used for shared state. Designing a more optimal cross-domain primitive is future work.

5 Low-Level Kernel Programming Idioms

Interface definition language KSplit IDL builds on the ideas from existing driver isolation projects [33, 62, 66]. Specifically, we borrow the idea of "projections" that describe the state synchronized across domains from LXDs [66] and extend them with rich IDL annotations that provide support for marshaling of low-level C idioms [33]. For every function crossing the boundary of isolated domains an IDL `rpc` declaration is generated. These declarations include projections for each composite type. Each projection is defined to include only the shared state fields that are read or written by the callee function, as determined by the parameter access analysis.
For ambiguous cases, additional annnotations are included for format (e.g., whether a pointer refers to a singleton or an array and type-specific formatting, such as null-termination) and size. KSplit aims to produce these annotations automatically

or warnings for programmers to address.

Pointer classification The main challenge for the static analysis is to infer IDL specifications for format and size to marshal data correctly from the low-level type information available in C. For each data type in a projection whose marshaling requirements are ambiguous, we leverage our PDG representation to compute: 1) the def-use chain and aliases for the references that can be assigned to the ambiguous argument to determine what kinds of operations may be performed on the argument (e.g., to distinguish singletons and arrays) and 2) all the call sites in which the ambiguous argument is used to infer semantics from uses (e.g., to infer strings from argument use in string manipulation functions).

KSplit uses this information to iteratively refine knowledge about the marshaling requirements of arguments, resolving the ambiguities in some cases and producing specific warnings in other cases. For example, suppose that an argument has the type `char *`, but we do not know whether this type refers to a singleton, an array, a null-terminated array (i.e., sentinel array), or another data type altogether (e.g., due to a type cast). KSplit resolves such an ambiguity by leveraging the def-use information of its aliases to classify the argument and apply further analyses to determine if the ambiguity can be resolved. For classification, we employ the CCured method [69] as implemented for LLVM in the NesCheck system [64]. CCured classifies pointers by whether they are involved in type casts (`wild`), are referenced using pointer arithmetic (`sequential`), or neither (`safe`). Pointers classified as safe by CCured/NesCheck are singletons, as these pointers reference only one location. Sequential pointers may be either arrays or structures, although these can be distinguished based on the way they are accessed. Finally, wild pointers involve type cast operations making their types ambiguous, although we can still infer type information in several cases for common patterns.

Once we have performed the classification, we then perform specialized analyses based on the class for further disambiguation:

Sized and null-terminated arrays KSplit can identify arrays whose size is determined at allocation time. It statically detects strings from uses of pointer aliases in any string manipulation functions.

Tagged and anonymous unions Deriving projections for union types is challenging: the type and union field name information is lost at the level of LLVM IR as compiler treats unions as raw bytes and simply accesses the fields as offsets. We develop an analysis algorithm that reconstructs field name information by matching offset accessed by the IR instructions with the offsets of each field. To marshal the union, the IDL compiler relies on a user-supplied discriminator function to determine the type of the union at runtime.

Recursive data structures KSplit supports marshaling of generic recursive data structures, e.g., linked lists, trees, and graphs. For example, to support a linked list, the static analysis generates a projection that includes a pointer to a projection of the same type as one of the fields. The marshaling code generated by the IDL compiler traverses all the pointers creating a map of visited objects until a fixed point is reached on cyclic graphs.

Opaque pointers and pointer errors If an argument is found to be wild, KSplit can resolve the type for some cases, including for void pointers cast to a single type [33]. KSplit handles some other common cases, such as the pattern where kernel APIs may return a reference to either an object or an error.

Other idioms KSplit is able to detect other special cases, such as buffers from user space, co-located data structures, and "container of"/"member of" data structures, to enable special handling (e.g., checks for user-defined memory) and targeted warnings (e.g., for marshaling ad hoc objects within a data structure).

6 Implementation

The KSplit system consists of a set of LLVM passes to perform the static analyses, an IDL compiler to generate synchronization code, and runtime support for isolation and tracking allocations and deallocations. The LLVM static analyses consist of 8,373 SLOC in C++ for PDG construction [59], shared data analysis (see Section 4.2), parameter access analysis (see Section 4.3), and atomic region analysis (see Section 4.4). PDG construction additionally uses the SVF alias analysis [79] for the intra-procedural alias analysis. We also use NesCheck [64] to classify pointers for resolving ambiguities in kernel idioms. LLVM analyses use bitcode produced using the O0 optimization level to preserve source semantics.

We implement KSplit for the LVDs framework that supports isolation of privileged kernel code through a combination of hardware-assisted virtualization and EPT switching [68]. Specifically, we rely on the LVDs execution environment to run the driver. We, however, implement a new IDL compiler to support synchronization between subsystems; LVDs supported synchronization of only basic types and data structures, but lacked support for arrays, unions, and recursive data structures. The compiler is implemented from scratch in 4,100 lines of C++.

Object lifetimes The main challenge of the runtime is to ensure that objects allocated or deallocated on one side of the isolation boundary are available or removed, respectively, from the other side. However, the tight integration of kernel and drivers historically has created irregular allocation and deallocation patterns. KSplit relies on a hybrid static and dynamic approach in which the execution runtime tracks new objects and allocates them each time a new object is passed across the isolation boundary. We, however, rely on static analysis to identify deallocation sites and instrument them to propagate deallocations across the isolation boundary.

	coretemp	nullnet	ixgbe	alx	can-raw	sb_edac	null_blk	dm_zero	msr	xhci-hcd
SLOC	562	194	27K	3K	615	2K	690	54	218	10K
Drv.→kern.	21	14	134	61	36	15	36	3	16	45
Kern.→drv.	2	11	81	26	17	1	9	2	5	27
Functions	643	1K	5K	3K	1K	912	1K	133	459	1K

(a) Complexity of driver analysis

Deep copy	31K	46K	999K	214K	153K	24K	75K	11K	24K	134K
Access analysis [33]	127	231	4K	1K	696	91	562	29	66	375
Shared analysis	87	156	3K	831	368	70	406	21	55	265
Boundary analysis	87	155	2K	806	333	70	379	21	51	194

(b) Total number of fields marshaled across all interface functions by each algorithm

Pointers	12K/76	19K/96	404K/1,529	84K/356	60K/178	9K/58	29K/220	4K/16	9K/44	51K/189
Unions	0/0	5/3	114/33	29/17	22/30	0/0	1/12	0/0	0/0	0/7
Critical sections	5/0	5/1	70/3	25/2	19/2	2/0	31/0	0/0	8/0	10/0
RCU	0/0	1/0	8/0	6/0	9/0	0/0	6/0	0/0	0/0	0/0
Seqlock	0/0	0/0	3/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Atomic operations	0/0	25/1	173/35	59/22	49/1	5/0	37/2	3/0	3/0	50/4
Container of	225/4	557/2	2K/20	1K/12	749/8	419/0	627/5	73/3	68/2	1K/6

(c) Impact of shared state optimizations (private/shared)

Singleton	70/0	84/0	1,251/0	310/0	147/0	39/0	183/0	15/0	41/0	172/0
Array	0/1	3/2	92/27	32/2	21/5	5/6	10/5	0/0	0/1	1/0
String	1/0	1/0	2/0	0/0	0/0	2/0	2/0	0/0	1/0	0/0
Wild pointer (void)	2/1	4/0	142/1	12/0	5/0	3/0	17/0	1/0	1/0	16/0
Wild pointer (other)	1/0	0/2	1/3	0/3	0/0	0/3	0/3	0/0	0/0	0/0

(d) Inference type semantics on shared pointers (handled/manual)

Time	17	217	546	190	135	22	490	5	7	238

(e) Analysis execution time (seconds)

Statements	70%	86%	50%	72%	79%	63%	79%	85%	77%	55%
Branches	57%	81%	48%	76%	79%	65%	91%	100%	96%	53%

(f) Test coverage

IDL (LOC)	163	221	2K	674	470	236	306	47	109	1K
IDL changes (LOC)	1	5	52	25	30	5	11	0	2	7
Drv. changes (LOC)	10	6	19	11	12	0	0	0	0	0
False positives	1	25	129	43	30	6	34	2	5	12
Ptr. misclassifications	0	0	7	3	2	2	3	0	2	0
Warnings	1	8	65	22	35	5	20	0	3	7

(g) Manual effort

Table 1: Driver complexity and impact of shared state optimizations.

7 Evaluation

To evaluate KSplit, we utilize CloudLab [73] c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.60 GHz, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. We use an Intel i7-4790K desktop for evaluation of `alx` network, `xhci` USB host-controller and Intel ME drivers. Both machines run 64-bit Ubuntu 18.04 Linux with kernel version 4.8.4.

7.1 Generality of Static Analysis

The main question is whether KSplit can be used as a general tool for the isolation of device drivers in the Linux kernel. To answer this question, we use the KSplit analysis to produce IDL for 354 drivers from multiple Linux subsystems (Table 2) and then evaluate the effectiveness of the analysis and IDL generation algorithms by isolating and validating the correctness of 10 drivers (Table 1). We chose a range of device and protocol drivers that represent typical kernel programming

	char/tty (77)	block (17)	net (89)	edac (13)	hwmon (67)	spi/i2c (38)	usb (53)
SLOC	1047	2535	13302	896	556	471	1340
Drv.→kern.	11	60	25	18	10	14	16
Kern.→drv.	10	16	47	4	5	3	13
Functions	546	2588	2691	839	462	772	784

(a) Complexity of driver interfaces

Pointers	15K /64	53K /310	73K /353	16K /107	10K /61	12K /71	18K /92
Unions	0/2	3/12	7/6	0/2	<1/<1	0/2	<1/4
Crit. sec.	5/<1	51/<1	25/<1	5/<1	6/<1	9/<1	9/<1
Atomic op.	<1/0	6/0	2/0	0/0	<1/0	<1/0	<1/<1
RCU	<1/0	<1/0	<1/0	0/0	<1/0	0/0	<1/<1
Seqlock	9/<1	45/2	45/11	6/0	<1/<1	4/0	10/<1
Container of	145/4	833/3	1K/9	338/2	133/2	207/2	215/3

(b) Impact of shared state optimizations (private/shared)

Singleton	53/0	26/0	303/0	84/0	56/0	66/0	81/0
Array	5/2	27/15	44/20	22/6	2/<1	4/2	4/1
String	<1/0	3/0	<1/0	2/0	<1/0	<1/0	<1/0
Wild (void)	5/<1	18/0	12/1	3/0	1/<1	2/<1	6/<1
Wild (other)	0/<1	0/2	0/3	0/3	0/<1	0/<1	0/2

(c) Inferred type semantics on shared pointers (handled/manual)

Table 2: Performance and complexity metrics across several subsystems (average per driver).

Reference	`ixgbe`		skx_edac
	nullnet	alx	sb_edac
Shared rpcs	11	73	13
Shared rpcs IDL$_\Delta$	+0/-51	+12/-29	+1/-1
Shared rpcs Annotat.$_\Delta$	0	+3/-3	0
New IDL	77	36	0

Table 3: Similarity within a class

and communication idioms: 1) `msr`: a high-level interface to the model specific registers (MSRs) on the Intel CPUs; it exercises several patterns typical for nearly every Linux device driver—dynamic registration of interfaces and callbacks, synchronization of null-terminated and statically sized arrays; 2) `nullnet`: a software-only network driver that emulates an infinitely fast network adapter; it relies on complex allocation of objects on both sides of the isolation boundary, and implements a fast data plane requiring careful handling of data structures to achieve optimal performance; 3) `coretemp`: temperature monitoring for CPU cores; it utilizes void pointers and two dimensional arrays; 4) `sb_edac`: error detection and correction (EDAC) for the Intel Skylake server integrated memory controllers; it requires marshaling of a graph of objects that describe the hierarchy of DRAM banks and memory controllers across the isolation boundary; 5) `null_blk`: a software-only emulation of the NVMe interface; it is similar to `nullnet`, which allows us to stress overheads of the isolation on a fast NVMe interface; 6) `ixgbe`: an Intel 82599 10Gbps Ethernet driver; it exhibits several critical characteristics interesting for decomposition: first, it relies on atomic operations to update packet statistics in the kernel; second, it exhibits a broad range of asynchronous accesses from system calls, interrupt contexts, software IRQs and New API (NAPI) threads that implement submission of packets and polling; third, it relies on system timers for several control plane operations that allow us to test static analysis for support of callback functions dynamically registered with the kernel; 7) `alx`: Linux Qualcomm Atheros ethernet driver; it is chosen to compare complexity and manual effort of decomposing device drivers within the same device class (i.e., we compare two ethernet drivers: `ixgbe` and `alx`); 8) `can_raw`: raw CAN protocol driver using the sockets API; it represents a protocol (i.e., not a device) driver and exhibits typical protocol layer patterns by interacting with the kernel network stack; 9) `dm_zero` is chosen to evaluate if we can fully automate (with no manual effort) for simple device drivers; 10) `xhci-hcd`: xHCI protocol driver for supporting USB 3.0; it handles complex interactions of the USB communication protocol as well as interacting with multiple kernel subsystems (PCIe, USB, and DMA).

By generating the IDL for a wide variety of drivers, we examine the generality of the KSplit analyses for producing IDL specifications and assessing the manual effort required for isolating many Linux kernel drivers. While we did not run all the 354 drivers, as we need the appropriate hardware, we compare metrics related to the effort of isolating an average driver to those we validated. To validate the 10 drivers, we perform the manual tasks required to complete the IDL as specified by the warnings generated by KSplit, and we perform static and runtime tests to determine the precision and accuracy of the KSplit static analysis.

Complexity of driver interfaces To justify the need for automated analysis techniques, we collect several metrics that illustrate the complexity of the 10 drivers isolated using KSplit (Table 1a). The two most complex drivers are `ixgbe` (over 27K SLOC) and `xhci` (over 10K SLOC). The `ixgbe` driver consists of over 2,000 functions, registers 81 callback functions with the kernel, and relies on 134 kernel functions for its operation. Isolation of the `ixgbe` driver involves analysis of 5,782 functions that may access the state shared between the kernel and the driver. A total of 999,136 fields and scalar arguments are transitively reachable from the arguments of driver functions that define its isolation boundary (Table 1b). While partial isolation of the `ixgbe` driver was demonstrated before [66, 68], isolation of the complete driver is beyond the reach of manual human analysis.

Impact of shared state optimizations KSplit distinguishes the shared state from the private state, which is critical for scalability of the analysis algorithms (Section 4.3). We collect the total number of fields in all data structures that are recursively reachable from all the arguments passed across the isolation boundary, i.e., previous approaches relied on naive "deep copy" [59] and field access approaches [33] (Table 1b). Out of 999,136 fields reachable through the isolation boundary of the `ixgbe` driver, only 4,509 fields are accessed, and an even smaller fraction of them, or 3,029, are shared (Table 1a). Furthermore, by reasoning about nested crossings of the isolation boundary, we reduce this number to 2,669. Most critically, the shared state optimization radically simplifies isolation of the driver, as in many cases complex low-level idioms, e.g., tagged unions, stay on one side of the isolation boundary (Table 1c). For example, out of 73 critical sections in `ixgbe`, only 3 are shared (`ixgbe` relies on the global `rtnl_lock` to register the driver with the kernel); all RCU and seqlocks are private, and do not trigger cross-boundary synchronization.

Pointer classification To understand how well KSplit supports classification of pointer references, we characterize the number of supported and problematic pointer patterns in our drivers (Table 1d). In many cases, KSplit is able to infer the types and sizes to enable IDL generation automatically. Table 1d shows that for `ixgbe`, out of 1,529 pointers (see the "Pointers" row in Table 1c) that require marshalling across the isolation boundary, only 31 require manual inspection to generate correct marshaling attributes. There are a small number of misclassified pointers as shown in Table 1g in the row of pointer misclassifications ("Ptr. Misclassifications"). We found that these misclassified pointers are sequential pointers that are wrongly classified as singleton pointers, because CCured fails to identify pointer-arithmetic operations on them. A detailed study of these misclassified pointers revealed the main reason for misclassification is due to not analyzing library code. For example, the ixgbe driver calls the kernel function `pci_request_selected_regions()` with a reference to the driver name string, but the kernel function itself does not perform pointer-arithmetic operations on the reference; instead it passes the reference to string library functions for processing the string. This causes CCured to misclassify the pointer as a singleton pointer. We could resolve many of these misclassification cases by manually encoding how pointers are used in library functions. For example, if a pointer is passed to logging functions such as `printk()` or other string library functions, e.g, `strcmp()`, we can classify the pointer as a sequential pointer.

Analysis execution time To understand practicality of KSplit and its ability to be part of the kernel development toolchain, we measure the execution time of the analysis (Table 1e). The execution time is largely influenced by the number of functions that are involved in the analysis (this number is determined primarily by the size of the driver and by the size of the kernel subsystem the driver interacts with). Complex device drivers that interact with multiple subsystems (e.g., `can_raw`, `null_blk`, `xhci`, and `ixgbe`), require 190-546 seconds to complete. Simple device drivers finish in under a minute.

Precision of the analysis and manual effort To understand the precision of the analysis and the manual effort involved in isolation of a driver, we compare an automatically generated IDL with the final IDL used for isolation of the driver. As we do not have the ground truth, to gain confidence in the correctness of the isolated driver, we execute a collection of tests on each driver. We use Gcov to collect the code coverage metrics for the tests we run (Table 1f). The code coverage is less than 50% in some cases as we can only trigger execution of a subset of the driver code (i.e., EDAC drivers support multiple generations of Intel CPUs from Ivy Bridge to Xeon Phi; `ixgbe` supports multiple hardware interfaces, e.g., x540, 82599, 82598; `xhci`, being a protocol driver, has a lot of error handling code, e.g., in a representative function `handle_tx_event()` that handles all the usb transmit events, out of 348 total source lines, 198 (or 56%) lines are error handling code that we cannot trigger without fault injection; `sb_edac` driver consists of 1162 lines of code, out of which only 492 (42%) are executable on our Haswell hardware, out of which our tests cover 373 lines of code (thus increasing our coverage from 63% to 76%).

The statistics for the automatically-generated IDL and the manual effort required in resolving warnings are shown in Table 1g. An IDL of a complex driver, like `ixgbe`, generated by KSplit, consists of 2,476 lines of code. Isolation of the driver required changing 53 lines of the automatically generated IDL (or 2% of the IDL). We only had to introduce 19 lines of changes to the code of the driver, which mostly involve redefinition of certain macros as helper functions (e.g., `setup_timer`, `INIT_WORK`, etc.). KSplit misclassified 7 out of 999 pointers shared across the isolation boundary. Two pointers were strings that were passed across the isolation boundary, but were not accessed through pointer arithmetic or string manipulation functions. One pointer was referring to a region of DMA'ed memory (again, not used in any pointer arithmetic). Four pointers were misclassified due to being passed as arguments to the `memcpy()` function. For smaller drivers, isolation requires less than 30 lines of IDL changes. Furthermore, most small drivers require no changes to the driver code.

The "False Positives" row indicates the number of fields identified by KSplit that are not found to be necessary based on our manual analysis and driver profiling. The ground truth may be incomplete, so this number represents an upper bound in the number of false positives. The fraction of false positives is generally low (<10%). The dominant reason for false positives are aliases in the shared data analyses (shared data uses a type-based approach that leads to overapproximation of fields and `in`/`out` attributes).

Finally, the "Warnings" row shows the number of warnings KSplit's static analyses generate for each driver. These warnings must be resolved by developers to obtain a working IDL.

Similarity within a class A key insight for isolation of a large fraction of the drivers in the kernel is based on the assumption that drivers within the same class have significant degree of similarity across their interfaces. Isolation of one driver within the class, therefore, could guide the isolation of other drivers in a relatively straightforward manner, hence amortizing manual effort across the class. To understand the

	Null	Integer	Array	String	Void	Union
Bytes	0	8	32 * 8	256	4096	24 + 32
Cycles	502	532	690	1310	919	710

Table 4: Overhead of marshaling various data structures

effort involved in isolating multiple drivers in the same class, we choose a base driver within a class and compare it with other drivers in the class (Table 3). For example, for network drivers we compare `alx` and `nullnet` to the base `ixgbe` driver. The `alx` driver shares 73 function definitions (`rpcs`) with `ixgbe` (the total number of functions crossing the isolation boundary in both directions is in Table 1a). After `ixgbe` was decomposed, decomposition of `alx` required changes to 6 annotations and a total 41 lines of changeset in the shared part of the IDL.

Generality of IDL generation To judge if KSplit can be used as an isolation tool for the entire population of drivers, we apply it to 354 drivers (Table 2) across nine subsystems in the Linux kernel. To make a prediction about the manual effort involved of isolation of an average driver, we collect the same metrics as the ones collected for the validated drivers (Table 1), although all the counts in Table 2 are averages per driver. In general, we see a huge impact due to the shared state optimizations (Table 2b) and a low number of problematic pointer instances (i.e., cases that are not "singletons") that could result in warnings (Table 2c). We therefore believe that the effort of isolating an average driver in these subsystems is comparable to the drivers we validated.

IDL warnings KSplit produces IDL warnings for the following patterns in Table 2c: 1) arrays (including "strings") with undetermined size; 2) wild pointers whose type cannot be inferred deterministically from "wild (void)"; 3) anonymous unions in "wild (other)"; and 4) potential cases of collocated data structures in "wild (other)". In general, the number of IDL warnings for each driver is dependent not only on the size of the driver, i.e., lines of code, and complexity of the driver interface, i.e., lines of IDL code, but also on the types of kernel idioms used for communication across the isolation boundary. For example, isolation of the `alx` driver involves an IDL file that consists of 674 lines of code and requires analysis of 22 warnings. The `alx` driver contains 17 anonymous unions, 2 undetermined size arrays and 3 non-void wild pointers. At the same time, isolation of the `can-raw` driver that uses a smaller IDL (470 lines of IDL code) yields 35 warnings. The high number of warnings for `can-raw` is attributed to the 30 instances of anonymous unions and 5 indeterminate-size arrays in its interface.

7.1.1 Case Study: Ixgbe Network Driver

To illustate the process of decomposition, we consider an example, the `ixgbe` driver, that combines a representative set of complex kernel data structures, low-level idioms, and synchronization patterns. As discussed above, separation of shared and private state is critical for reducing complexity of the IDL required for isolation of `ixgbe`. KSplit automatically resolves

```
projection<struct sk_buff> skb_xmit {
  projection net_device *dev;
  unsigned int len;
  unsigned int data_len;
  ...
  void * [alloc_sized<callee>(self->true_size)] head;
  void * [within<self->head, self->true_size>] data;
  unsigned int [within<_, self->true_size>] tail;
  unsigned int [within<_, self->true_size>] end;
};
```

Listing 1: Projection of an `sk_buff` data structure

all function pointers that `ixgbe` registers with the kernel as its interface, identifies five "user" and "ioremap" memory regions used by the interfaces of the driver, and `ixgbe` exchanges 119 opaque pointers across the isolation boundary, where only one requires manual intervention. `ixgbe` uses one function that returns a pointer-as-error, which is successfully identified by KSplit.

One of the most challenging parts of isolation is the proper handling of the `sk_buff` data structure, representing a network packet (Listing 1). Several integer fields are used as offsets into the data: 1) `tail` – to mark the end of the packet's data, and 2) `end` to represent the start of the `skb_shinfo` structure. The low-level PDG representation of the program allows us to derive that the `skb_shinfo` data structure is allocated within the `data` object. As the `tail` and `end` fields participate in pointer arithmetic operations, KSplit generates a special `bounds` IDL attribute that instructs the marshaling code to check that the field is within a specific range, but these bounds have to be specified manually.

KSplit support for recursive data structures allows us to marshal `sk_buff` buffers that consist of multiple fragments (`sk_buff` contains an optional list of fragments).

7.2 Performance

In general, the performance of the isolated driver is largely determined by the performance of an underlying isolation framework, i.e., LVDs in our current implementation [68]. We, however, quantify the impact of the KSplit-specific marshaling protocol, and conduct an end-to-end performance measurement of an application running on top of the isolated Ixgbe driver.

Marshaling overheads We perform microbenchmarks to evaluate the overheads of marshaling various data structures that are commonly used in the Linux kernel (Table 4). For each data structure, the test involves marshaling the data structure, passing it across the isolation boundary, and unmarshaling it. We perform ten million iterations and report an average. On the LVDs system, a null call-reply invocation takes 502 cycles which includes the overhead of executing the `vmfunc` instruction, saving and restoring general registers, and picking a stack inside the driver domain. KSplit adds 30 cycles for marshaling simple scalar fields such as integers. For marshaling tagged unions, we rely on a user-supplied discriminator function that identifies the tag and marshals the union accord-

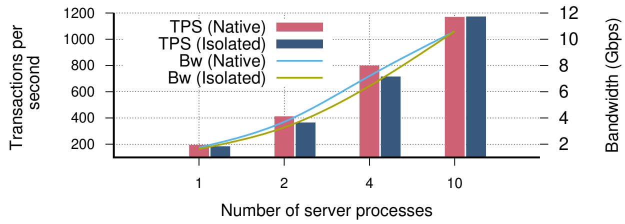

Figure 4: Memcached performance

ing to the type it represents. In our experiment, we marshal a union that represents a string of 32 characters, which incurs an overhead of 208 cycles.

Memcached To understand end-to-end overheads of isolation on real application workloads, we utilize an experiment that runs memcached, a high-performance, in-memory object caching system [4] and compare a native, non-isolated kernel with the performance of a system that utilizes an isolated version of the Ixgbe network driver. We run memcached version 1.5.12 with a single service thread and a cache size of 5GB. We use the memaslap [2] load-generator to send random UDP requests of 64B keys and 1024B values to the server (90% get and 10% set) with a concurrency of 128. To make a fair comparison, we limit the number of available cores to 10, as we are limited by the performance of a 10Gbps adapter (all 20 cores would allow isolated drivers to bridge the performance gap but at a cost of a higher CPU utilization). We report both the number of key-value transactions per second and total network bandwidth (Figure 4). For experiments with 1-4 threads, KSplit stays within 5.4-18.7% of the non-isolated system's performance. With 10 threads, both isolated and native drivers saturate the network interface and hence demonstrate a nearly identical performance (albeit at a higher CPU utilization due to domain crossings).

8 Conclusions

After decades of research, commodity CPUs are converging on a set of practical hardware mechanisms capable of providing support for low-overhead isolation. With the performance no longer being the main roadblock, complexity becomes the main challenge for enabling isolation in commodity systems. Our work on KSplit takes a step forward by enabling isolation of unmodified device drivers in the Linux kernel. A combination of practical static analysis techniques allows us to address the daunting complexity of the driver interfaces—KSplit supports isolation of complex, fully-featured device drivers with only minimal changes and human involvement. While our current implementation works with Linux and a specific isolation framework, we argue that our analysis and state-synchronization techniques are general and can serve as a foundation for a range of isolation solutions enabled by the emerging hardware mechanisms.

Acknowledgments

We thank OSDI'21, SOSP'21 and OSDI'22 reviewers and our shepherd, Rüdiger Kapitza, for in-depth feedback on earlier versions of the paper. We would like to thank the Utah CloudLab team for continuous support in accommodating our hardware requests. Finally, we would like to thank the artifact evaluation committee for numerous comments that greatly improved the artifact. This research is supported in part by the National Science Foundation under Grant Numbers CNS-1527526, OAC-1840197, CNS-1801534, CNS-1816282, and DARPA HR0011-19-C-0106. Vikram Narayanan is partly supported by an IBM PhD fellowship.

References

[1] Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[2] libmemcached. https://libmemcached.org/libMemcached.html.

[3] LKDDb: Linux Kernel Driver DataBase. https://cateee.net/lkddb/. Accessed on 04.23.2019.

[4] Memcached. https://memcached.org/.

[5] PKS: Add protection keys supervisor (PKS) support. https://lwn.net/Articles/826091/.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[7] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, page 175–188, 2016.

[8] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing linux kernel components in K42. Technical report, IBM Watson Research, 2002.

[9] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for

scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, 2009.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, page 267–283, 1995.

[12] D. W. Boettner and M. T. Alexander. The Michigan Terminal System. *Proceedings of the IEEE*, 63(6):912–918, June 1975.

[13] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[14] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *2010 USENIX Annual Technical Conference (USENIX ATC '10)*, 2010.

[15] Bromium. Bromium micro-virtualization, 2010. `http://www.bromium.com/misc/BromiumMicrovirtualization.pdf`.

[16] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th Usenix Security Symposium*, pages 57–72, 2004.

[17] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[18] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 45–58, 2009.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[20] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, page 18–37, 2015.

[21] Stephen Chong, Jed Liu, Andrew Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web Applications via Automatic Partitioning. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 31–44, 2007.

[22] DDEKit and DDE for linux. `http://os.inf.tu-dresden.de/ddekit/`.

[23] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pages 321–332, 2013.

[24] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 256–269. Springer, 2004.

[25] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88, 2006.

[26] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[27] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. Technical Report TUD-FI06-07, 2006.

[28] Flux Research Group. CloudLab Web site. `http://www.cloudlab.us`.

[29] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 38–51, 1997.

[30] Alessandro Forin, David Golub, and Brian N Bershad. An I/O system for Mach 3.0. Carnegie-Mellon University. Department of Computer Science, 1991.

[31] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[32] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA '06)*, 2006.

[33] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178, 2008.

[34] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 193–206, 2003.

[35] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114, 2000.

[36] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 65–74, 1996.

[37] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC '02)*, page 45–58, 2002.

[38] David B Golub, Guy G Sotomayor, and Freeman L Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX MACH III Symposium*, pages 153–172, 1993.

[39] Google. Fuchsia project. https://fuchsia.dev/fuchsia-src/getting_started.md.

[40] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 401–417, 2020.

[41] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 1016–1031, 2015.

[42] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical report, TU Dresden, Dresden, Germany, 2003.

[43] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 489–504, 2019.

[44] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[45] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[46] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[47] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. Epti: Efficient defence against meltdown attack for unpatched vms. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 255–266, 2018.

[48] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[49] INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[50] Trent Jaeger. *Operating System Security*. Morgan & Claypool, 2008.

[51] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 473–484, 2013.

[52] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. Doctoral thesis, School of Science, 2012.

[53] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware, 2017.

[54] C. Lattner, A. Lanharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[55] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI '04)*, pages 17–30, 2004.

[56] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 234–251, 2017.

[57] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two Years of Experience with a μ-Kernel Based OS. *ACM SIGOPS Operating Systems Review*, 25(2):51–62, April 1991.

[58] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 285–298, 2017.

[59] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2359–2371, 2017.

[60] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative Privilege Separation. In *26th ACM Conference on Computer and Communications Security (CCS '19)*, pages 1023–1040, 2019.

[61] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 1607–1619, 2015.

[62] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 115–128, 2011.

[63] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*, pages 1–15, 2019.

[64] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory Safety for Embedded Devices with NesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*, page 127–139, 2017.

[65] Brendan Murphy. Automating Software Failure Reporting: We Can Only Fix Those Bugs We Know About. *Queue*, 2(8):42–48, November 2004.

[66] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs : Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[67] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 21–39, 2020.

[68] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, page 157–171, 2020.

[69] George Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[70] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 116–132, 2013.

[71] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[72] Matthew J Renzelmann and Michael M Swift. Decaf: Moving Device Drivers to a Modern Language. In *2009 USENIX Annual Technical Conference (USENIX ATC '09)*, 2009.

[73] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[74] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated Partitioning of Android Applications for Trusted Execution Environments.

In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 923–934, 2016.

[75] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[76] Leonid Ryzhyk. *On the Construction of Reliable Device Drivers*. PhD thesis, UNSW, January 2010.

[77] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, page 275–288, 2009.

[78] Jerome Saltzer and Michael Schroeder. The protection of information in computer systems. *Proceedings of The IEEE*, 63(9):1278–1308, September 1975.

[79] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266, 2016.

[80] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[81] Hajime Tazaki. An introduction of library operating system for linux (LibOS). `https://lwn.net/Articles/637658/`.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, 2019.

[83] Kevin Thomas Van Maren. The Fluke device driver framework. Master's thesis, The University of Utah, 1999.

[84] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[85] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, 1993.

[86] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 241–254, 2008.

[87] Yang Liu Yongzheng Wu, Jun Sun and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 323–333, 2013.

[88] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS)*, 20(3):283–328, 2002.

[89] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 236–250, 2003.

A Artifact Appendix

Abstract

We release the source code of all software used in this paper along with detailed build instructions and automated scripts used for running the benchmarks as a collection of publicly-hosted Git repositories.

Scope

The artifact allows one to run static analysis on the set of drivers we isolated for this paper and collect metrics that are reported in Table 1, Table 2, and Table 4.

Contents

The artifact consists of the source code for the following subsystems: 1) KSplit analysis framework used to generate interface definition language (IDL) files `https://github.com/ksplit/pdg`; 2) LLVM bitcode files for the drivers analyzed in the paper `https://github.com/ksplit/bc-files` (we provide detailed instructions for how to re-generate the bitcode files, however, to simplify the process of re-creating results reported in the paper, we provide a collection of pre-generated files); 3) KSplit IDL compiler that generates the glue code required to execute the driver in isolation from the IDL files `https://github.com/ksplit/idlc`; 4) a modified Linux kernel that executes isolated drivers in Lightweight Virtualized Domains (LVDs) [68] `https://github.com/ksplit/lvd-linux`; and 5) a modified Bareflank hypervisor that provides secure and efficient isolation boundary based on VMFUNC EPT switching interface used by LVDs `https://github.com/ksplit/bflank`.

Hosting

The artifact is hosted on GitHub. The `README.md` file under `https://github.com/ksplit/ksplit-artifacts` details the steps required to build and run the benchmarks.

We conduct all experiments in the openly-available CloudLab cloud infrastructure testbed [28] and make our experimentation environment available via an open CloudLab [73] profile that automatically instantiates the software setup required to run KSplit: `https://github.com/ksplit/ksplit-cloudlab/`.

Requirements

The KSplit build infrastructure was tested on an x86-64 Ubuntu 18.04 LTS system. The static analysis framework is built and tested against LLVM v10.0.1. We rely on LVDs [68] to execute isolated drivers. LVDs run on any modern Intel x86-64 hardware (Haswell or later) that supports virtualization (Intel VT-x) and EPTP switching via VMFUNC. LVDs rely on a customized Bareflank hypervisor and a modified Linux kernel based on v4.8.4. We have tested KSplit on the following hardware (available in CloudLab): a Cisco UCS C220 machine configured with an Intel Xeon E5-2660 CPU, and a Dell PowerEdge C6420 machine configured with an Intel Xeon Gold 6142 CPU.