KSplit: Automating Device Driver Isolation
Yongzhe Huang, Pennsylvania State University; Vikram Narayanan and David Detweiler, University of California, Irvine; Kaiming Huang, Gang Tan, and Trent Jaeger, Pennsylvania State University; Anton Burtsev, University of California, Irvine, and University of Utah
https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe
KSsplit: Automating Device Driver Isolation

Yongzhe Huang∗1, Vikram Narayanan∗2, David Detweiler2, Kaiming Huang1, Gang Tan1, Trent Jaeger1, and Anton Burtsev2,3

1Pennsylvania State University
2University of California, Irvine
3University 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 KSsplit, a new framework for isolating unmodified device drivers in a modern, full-featured kernel. KSsplit 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 this shared state for efficient isolation. While some kernel idioms present ambiguities that cannot be resolved automatically at present, KSsplit classifies most ambiguous pointers and identifies ones requiring manual intervention. We evaluate our solution on nine subsystems in the Linux kernel by applying KSsplit to 354 device drivers and validating isolation for 10 drivers. For example, for a complex ixcgbe driver, KSsplit 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 KSsplit analysis of the 354 drivers shows a similar fraction of manual work is expected, showing that KSsplit 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 software fault isolation (SFI) implementations [53], will reduce isolation overheads even more.

Despite the 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 the shared memory environment of a monolithic kernel, where they freely exchange references to large and complex data structures (e.g., hierarchical, cyclic data structures with many data and pointer fields) 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. Moreover, 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, gradually 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, ixcgbe, 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 a subset of driver functionality that could be efficiently executed in isolation. As a result, many challenging parts of the drivers, e.g., interrupt handlers and optimized data-plane functions, remained inside the unisolated kernel. In addition, these techniques did not aim to minimize data synchronization overhead and required 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 used 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, which we address in the design of KSplit. 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 more accurate analysis methods to be targeted to the relevant subset of the kernel.

Second, modern drivers execute in a concurrent, fully-reentrant environment of the kernel, 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-terminated and sized arrays, tagged and anonymous unions, self-referential data structures like linked lists, etc. To separate complete drivers, 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 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 19 lines of driver code changes and only 53 lines of IDL changes for the 2,476 lines of device interface definitions. We also apply KSplit 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, retaining 5.4–18.7% of the non-isolated system’s 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, extended page table (EPT) switching),
core-isolation [66], or techniques of software fault isolation (SFI) [18, 25, 62, 85] that enforce a segment-like isolation boundary around the driver code through instrumentation of control flow and memory instructions.

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 packet buffers, 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, these 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”, which 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 analysis 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-performance-critical 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 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]. 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].

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 to improve boundary-crossing performance, providing an interface definition language (IDL) for specifying which data requires synchronization in driver interfaces. This work demonstrates the potential for the efficient hardware-based protection domain isolation of legacy drivers. However, such isolation required a significant manual effort to develop 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 with 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 must be synchronized on invocations that cross the isolation boundary, and also at the points where the driver uses concurrency primitives, e.g., atomics, spinlocks, mutexes, ready-copy-update (RCU), etc.

For example, the kernel submits a network packet to a network device with the ndo_start_xmit() function:

```
1  ndo_start_xmit(struct sk_buff *skb,
2     struct net_device *netdev)
```

KSplit ensures that all the fields shared between the kernel and driver of all the data structures that are recursively reachable from the two input arguments (i.e., skb and netdev), 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

---

2To 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.
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 up-to-date copies in both domains.

KSplit provides software analysis algorithms that 1) compute the subset of the kernel state that is accessed by the driver (i.e., the shared state) and 2) synchronize the shared state on cross-domain invocations and on the use of 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 that 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 (Figure 1a). 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 (Figure 1b). For example, some sk_buff pointers are used for in-place access to parts of the network packet, i.e., head and data mark the beginning of the packet header, and the data regions from which the packet is assembled, respectively.

To compute shared state accurately under these requirements, KSplit employs a field-sensitive data-flow analysis using a modular 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 directly 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 up-to-date shared state, regardless of how the kernel and driver interact. The kernel, however, invokes functions of the driver in parallel on multiple CPUs. Moreover, 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 driver registration.

KSplit leverages the 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 Kernel code utilizes a range of low-level idioms that create ambiguities for static analysis (Figure 2). For example, device drivers rely on sentinel values (e.g., NULL) to represent variable-sized arrays, e.g., the PCI subsystem uses the pci_id_table array to store a set of devices supported by a particular driver (Figure 2a). To optimize allocation and
deallocation of objects, kernel can collocate multiple data structures into one memory area, and use pointer arithmetic to access them (Figure 2b). 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, require special treatment, when passed into an isolated driver, e.g., marked as allocated in user memory with the user attribute (Figure 2c). While recursive data structures are rarely passed across the kernel-driver interface, some drivers use linked lists, and even generic graphs of recursive objects (Figure 2d). Tagged and anonymous unions are used by the driver to implement polymorphic functions that can take generic arguments of a union type (Figure 2e). Device drivers frequently rely on void* pointers to express type polymorphism (Figure 2f). Another typical pattern for the kernel APIs is to return an error as a specially formed pointer—this allows a simple unified function signature, e.g., the struct request *blk.mq.allocation() function from the block driver returns a pointer to the block request on successful invocation, but can return a specially formed pointer that represents an error otherwise. 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 the annotations to resolve ambiguities in marshaling manually for most cases [33, 35, 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 ones. For example, char * references, such as 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 caones 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 methods aimed at the 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, and supported only a limited subset of kernel idioms. Ksplit leverages advances in static analysis: specifically, a combination of an accurate program dependence graph (PDG) 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 marshaling of data in 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]. Specifically, 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 defects, 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 ones. For example, char * references, such as 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 methods aimed at the 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, and supported only a limited subset of kernel idioms. Ksplit leverages advances in static analysis: specifically, a combination of an accurate program dependence graph (PDG) 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 marshaling of data in the low-level driver code.
Finally, speculative execution and side-channel attacks on each boundary crossing for that shared data; and (4) 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 the general case, 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 most cases. While some of these idioms are commonly applied in C programs, some idioms may need to be replaced for other kernels.

4 KSplit Static Analysis

Figure 3 presents KSplit analysis 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 frontend. 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 the general case, 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 most 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 dependencies between 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 then 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 the boundary.

Scaling alias analysis with parameter trees A common type of data dependence happens when an instruction writes to a memory region from which another instruction reads. Computing such memory-related data dependences requires alias analysis, which computes the variables or expressions that may reference (i.e., point to) the same memory object, and are called aliases. We must compute aliases in KSplit because we must detect all objects that may be accessed by both the kernel and the drivers. Further, the isolation of the driver code requires an interprocedural alias analysis, as both the kernel and driver code may pass pointers to data objects through function calls, as well as through global variables. The alias analysis problem is known to be undecidable; devising a precise analysis that is scales well and is guaranteed to capture all aliases 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 then propagate those alias results inter-procedurally using the parameter tree approach [59]. This allows us to efficiently compute memory dependencies across function boundaries in a context-insensitive way. Specifically, it first constructs PDGs for each function in the program (which includes interprocedural memory dependencies) and then glues them together by connecting actual parameter trees for arguments at function call sites with formal parameter trees for parameters; details can be found in [59].

To illustrate the idea of parameter trees, consider the \texttt{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 4 shows a parameter tree for two arguments of the \texttt{msr.read()} function: 1) the \texttt{file} argument of type \texttt{struct file*} and 2) the \texttt{int} argument \texttt{count}. The parameter tree for the \texttt{file} argument has a root node labeled “\texttt{file:struct file*}” for representing the storage for the pointer, and a child node labeled “+\texttt{file:struct file}” for the memory region that the pointer points to. The references of each storage location in the pro-
4.2 Computing Shared and Private Data

Accurate separation of shared and private state is critical for the 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 neither the driver nor 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. This analysis leverages the modular alias analysis described 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 the 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 field accesses for that type. (4) For each field, if the field has accesses from both the kernel and the driver, we consider the field to be 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 are only accessed in the kernel. Our algorithm determines that features is shared, while the other two fields are private to the kernel.

This algorithm relies on two assumptions. First, in step (2), we assume that any state shared between the kernel and driver is accessed using one of the shared struct types from step (1). While this is not guaranteed, the kernel generally obeys typing for the types it shares with the drivers. If we miss a data type, we may under-approximate shared state, causing correctness issues, but we have not found any violations so far. 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 one instance of a given type to be shared while a different instance being private: e.g., if a device driver accesses the inode field of the struct file * object, it is typical 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 required by the callee domain to execute the call. Similarly, when the function returns, the changes the callee made to any shared state must be synchronized back to the caller, to reflect updates on its copy. We develop parameter access analysis to compute all data structures and 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 fixed point (when the worklist becomes empty). Algorithm 1 computes field usage that is only dependent on parameters passed between domains. 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
1 initialize AM to be empty
2 worklist ← \{f\}
3 while worklist is not empty do
   4 \( f_j \) ← remove_any(worklist)
   5 for node n in T do
      6 for instruction i in \( f_j \) do
         7 if G has a dependence edge from i to n then
            8 \( AL \) ← the edge’s access label
            9 \( AM[n] \) ← \( AM[n] \) ∪ \( AL \)
         10 else if i calls \( f_j \) then
            11 worklist ← worklist ∪ \{\( f_j \)\}
   12 end
   13 end
   14 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 \( f \_data \) structure, we associate a READ label with the parameter tree node that represents the storage of that \( f \_inode \). 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 identifies the 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 \( t \_d1 \) and \( k ' \) reads shared field \( t \_d2 \). The previous analysis determines both \( t \_d1 \) and \( t \_d2 \) need to be sent when \( k \) calls \( d \). However, our optimization distinguishes the two reads and sends only \( t \_d1 \).

### 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. The kernel and drivers synchronize access to the shared state through a variety of kernel-provided concurrency mechanisms: 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 into the kernel 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 atomicty across isolated domains, we rely on a 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) identify 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 either 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 the 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 acquire and 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 data reachable through a pointer used in one of the RCU update primitives, 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 negates the benefits of RCU locks, they are rarely used across the isolation boundary. 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”, which 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 an isolated domain, an IDL remote procedure call declaration is generated.

```c
1  rpc netdev_t _tx_t ixgbe_xmit_frame(
2    projection sk_buff [alloc(callee)] skb,
3    projection net_device *netdev)
```

For each argument of a composite type, e.g., struct, union, the IDL includes a projection that lists the shared-state fields that are read or written by the callee function, as determined by the parameter-access analysis (see the example projection for struct sk_buff in Section 7.1.1). For ambiguous cases, additional annotations are included to specify type of the object and in-memory representation (e.g., whether a pointer refers to a singleton or an array, and also type-specific formatting, such as null-termination) and size. KSplit aims to produce these annotations automatically, or generate warnings for programmers to address.

Pointer classification The main challenge for the static analysis is to infer IDL specifications from the low-level type information available in C. For each field type in a projection whose marshaling requirements are ambiguous, we leverage our PDG representation to compute: 1) aliases and def-use chains for references to the ambiguous argument, in order to determine what kinds of operations may be performed on it (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 the argument’s 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 others. 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., a sentinel array), or another data type altogether (e.g., due to a type cast). KSplit resolves such ambiguities by first leveraging the def-use information of the argument’s aliases and then refining the knowledge by applying further analyses. 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, which make 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 pointer 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: types and named of the union’s fields are lost at the level of LLVM IR, as the 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 the offsets 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 union’s active field 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.

Opaque pointers and error pointers If an argument is found to be wild, KSplit can resolve the type in some cases, e.g., 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 allocated in user space, co-located data struc-
tures, and “container-of”/“member-of” data structures, to enable special handling (e.g., marshaling of user memory) and targeted warnings (e.g., for marshaling objects collocated within a memory region or a data structure).

6 Implementation

The KSplit system consists of a set of LLVM passes to perform the static analysis, an IDL compiler to generate synchronization code, and a runtime component to track the allocation and deallocation of objects. The LLVM static analysis passes consist of 8,373 SLOC in C++ for PDG construction [59], shared data analysis (Section 4.2), parameter access analysis (Section 4.3), and atomic region analysis (Section 4.4). PDG construction additionally uses the SVF framework [79] for performing the intra-procedural alias analysis. We also use NesCheck [64] for pointer classification required to resolve ambiguities in kernel idioms. To preserve the source semantics, we use optimization level 0 to generate the LLVM bitcode.

We implement KSplit for the LVDs framework, which 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 isolated drivers. 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 for the runtime is to ensure that object allocation and deallocation on one side of the isolation boundary is reflected on the other side. Tight integration of the kernel and drivers has historically 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.

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 the 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 and communication idioms: 1) msr: a high-level interface to the model specific registers (MSRs) on the Intel CPUs, which 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, which 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, which utilizes void pointers and two-dimensional arrays; 4) sb.edac: error detection and correction (EDAC) for the Intel Skylake server integrated memory controllers, which 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; the driver is similar to nullnet, i.e., it allows us to stress-test the overheads of isolation on a fast NVMe interface; 6) ixgbe: an Intel 82599 10Gbps Ethernet driver, which exhibits several critical characteristics that are 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: a Linux Qualcomm Atheros ethernet driver; we select alx to compare the complexity and manual effort of decomposing device drivers within the same device class (i.e., we compare three ethernet drivers: ixgbe, alx, and nullnet); 8) can.raw: a raw CAN protocol driver using the sockets API, which represents a class of protocol drivers that exhibit typical protocol-layer patterns by interacting with the kernel network stack; 9) dm.zero: a software block driver that returns 0 on reads and drops writes; dm.zero tests if KSplit can fully automate isolation of simple device drivers; 10) xhci-hcd: an xHCI protocol driver for supporting USB 3.0, which handles complex interactions of the USB communication protocol.

A wide variety of drivers allows us to examine the generality of the KSplit analyses for producing IDL specifications, and assess the manual effort required for isolation. While we did not run all 354 drivers, we compare metrics related to the effort of isolating an average driver to those we validated. To validate the 10 drivers, we first perform manual tasks required to complete the IDL by resolving all warnings generated by KSplit, and then test the isolated driver trying to evaluate cor-
rectness of isolation what allows us to judge precision and accuracy of the 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 the 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 the 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 the scalability of the analysis algorithms (Section 4.3). We
Table 1: Performance and complexity metrics across several subsystems (average per driver).

<table>
<thead>
<tr>
<th></th>
<th>ixgbe</th>
<th>skb_edac</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC</td>
<td>1047</td>
<td>2535</td>
</tr>
<tr>
<td>block</td>
<td>13302</td>
<td>896</td>
</tr>
<tr>
<td>ethernet</td>
<td>355</td>
<td>556</td>
</tr>
<tr>
<td>iwd</td>
<td>471</td>
<td>471</td>
</tr>
<tr>
<td>span2c (kB)</td>
<td>1340</td>
<td>1340</td>
</tr>
</tbody>
</table>

(a) Complexity of driver interfaces

<table>
<thead>
<tr>
<th>Pointers</th>
<th>15K</th>
<th>33K</th>
<th>73K</th>
<th>16K</th>
<th>10K</th>
<th>12K</th>
<th>18K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unions</td>
<td>0/1</td>
<td>3/12</td>
<td>7/6</td>
<td>0/2</td>
<td>&lt;1/1</td>
<td>0/2</td>
<td>&lt;1/4</td>
</tr>
<tr>
<td>Crit. sec.</td>
<td>5/6</td>
<td>&lt;1/1</td>
<td>5/4</td>
<td>&lt;1/1</td>
<td>6/6</td>
<td>&lt;1/1</td>
<td>6/6</td>
</tr>
<tr>
<td>Atomic op.</td>
<td>&lt;1/0</td>
<td>6/6</td>
<td>2/0</td>
<td>2/0</td>
<td>2/0</td>
<td>2/0</td>
<td>2/0</td>
</tr>
<tr>
<td>RCU</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
<td>&lt;1/0</td>
</tr>
<tr>
<td>Seqlock</td>
<td>9/6</td>
<td>45/6</td>
<td>45/9</td>
<td>45/9</td>
<td>45/9</td>
<td>45/9</td>
<td>45/9</td>
</tr>
<tr>
<td>Container</td>
<td>145/4</td>
<td>833/3</td>
<td>1K9</td>
<td>338/2</td>
<td>1332/2</td>
<td>207/2</td>
<td>215/3</td>
</tr>
</tbody>
</table>

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

| Singleton | 53/0 | 26/0 | 303/0| 84/0 | 66/0 | 81/0 |
| Array     | 5/2  | 27/15| 44/20| 22/6 | 9/2  | 4/2  |
| String    | <1/0| <1/0| <1/0| <1/0| <1/0| <1/0| <1/0|
| Wild (void)| 5/1 | 18/0 | 121/0| 50/0 | 1/2  | 6/1  |
| Wild (other)| 0/1 | 0/2  | 0/3  | 0/3 | 0/1 | 0/2  |

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

Table 2: Similarity within a class

<table>
<thead>
<tr>
<th>Reference</th>
<th>ixgbe</th>
<th>skb_edac</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullnet</td>
<td>11</td>
<td>73</td>
</tr>
<tr>
<td>alx</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>sb_edac</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Similarity within a class

collect the total number of fields in all data structures that are recursively reachable from all the arguments passed across the isolation boundary—previous approaches relied on naive “deep copy” [59] and field-access approaches [33] (Table 1b). Out of 999K 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 the isolation of the driver, as in many cases, complex low-level idioms, e.g., tagged unions, stay on only 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 the 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 automatic IDL generation. Table 1d shows that for ixgbe, out of 1,529 pointers (“Pointers” in Table 1c) that require marshalling across the isolation boundary, only 31 require manual inspection to generate correct marshalling attributes. There is a small number of misclassified pointers (“Ptr. misclassifications” in Table 1g). We found that these misclassified pointers are sequential pointers that are wrongly classified as singleton pointers; 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_selectedregions() 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 a string library. This causes CCured to misclassify the pointer as a singleton pointer. It is possible to resolve some misclassification cases by either extending our analysis to kernel libraries (note, some library functions like printk() are challenging for static analysis), and by manually annotating how pointers are used in such functions. For example, if a pointer is passed to a string manipulation function, e.g., strncmp(), we can classify the pointer as sequential.

**Analysis execution time** To understand the practicality of KSplit and its fit for 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 the isolation of a driver, we compare an automatically-generated IDL with the final, manually-checked and tested IDL used for the 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, since we can only trigger the execution of a subset of the driver code. For example, 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 source lines, 198 lines (or 56%) are error handling code that we cannot trigger without fault injection; sb_edac driver consists of 1162 lines of code, 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%).

We collect several metrics that characterize the amount of manual effort involved in resolving IDL warnings (Table 1g). The 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 automatically-generated IDL (or 2% of the IDL). We only had to introduce 19 lines of changes to the driver’s code, which mostly involve redefinition of certain macros as helper functions (e.g., setup_timer, INIT_WORK, etc.). KSplit misclassified 7 out of 1,529 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 DMA memory region that was only accessed by the device but was not involved in any pointer arithmetic in the driver. Four pointers were misclassified due to being passed as arguments to the memcpy() function. For smaller drivers, isolation required less than 30 lines of IDL changes. Furthermore, most small drivers required no changes to the driver code.

The “False positives” row lists the number of fields falsely classified by KSplit as shared. We identify them as not shared through manual inspection and driver profiling. The ground truth may be incomplete, so this number represents an upper bound on 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 the overapproximation of fields and inout 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 scaling the isolation to a large fraction of all kernel drivers is grounded on the assumption that drivers within the same class have a 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 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 its class (Table 3). We compare two network drivers, alx and nullnet, to the base ixgbe driver. The alx driver shares 73 function definitions 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 changes 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 across nine subsystems in the Linux kernel (Table 2). To make a prediction about the manual effort involved in isolation of the average driver, we collect

<table>
<thead>
<tr>
<th></th>
<th>Null</th>
<th>Integer</th>
<th>Array</th>
<th>String</th>
<th>Void</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>0</td>
<td>8</td>
<td>32 * 8</td>
<td>256</td>
<td>4096</td>
<td>24 + 32</td>
</tr>
<tr>
<td>Cycles</td>
<td>502</td>
<td>532</td>
<td>690</td>
<td>1310</td>
<td>919</td>
<td>710</td>
</tr>
</tbody>
</table>

Table 4: Overhead of marshaling various data structures

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 and “strings” of 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 illustrate 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 the complexity of the IDL required for the isolation of ixgbe. KSplit automatically resolves 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. Out of 143 wild void pointers that ixgbe exchanges across the isolation boundary only one required manual intervention (“Wild pointer (void)” in Table 1d). We then had to inspect 3 wild pointers that are type casted between non-void types (“Wild pointer (other)” in Table 1d). The driver requires manual inspection of 27 array pointers (out of 119 exchanged across the boundary). The driver uses one function that returns a pointer-as-error, which is successfully identified by KSplit.

One of the most challenging parts of the ixgbe interface is the proper handling of the skb data structure, representing a network packet (Figure 1). Several integer fields are used
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 ensure 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 higher CPU utilization). We report both the number of key-value transactions per second and total network bandwidth (Figure 5). 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 nearly identical performance (albeit at 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 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 or 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 the ASPLOS’21, 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 continual 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.

Listing 1: Projection of an sk_buff data structure

```c
1  projection<struct sk_buff> skb_xmit {
2    projection net_device *dev;
3    unsigned int len;
4    unsigned int data_len;
5    ...
6    void * [alloc_sized<callee>(self->true_size)] head;
7    void * [within<self->head, self->true_size>] data;
8    unsigned int [within<_, self->true_size>] tail;
9    unsigned int [within<_, self->true_size>] end;
10  };
```

as offsets into the data: 1) tail marks the end of the packet’s data, and 2) end represents the start of the struct skb_shinfo that is collocated inside the data memory. 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 within IDL attribute that instructs the marshaling code to check that the field is within a specific range, but the range has to be specified manually (Listing 1). KSplit’s support for recursive data structures allows us to marsha 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 the underlying isolation framework, i.e., LVDS, in our current implementation [68]. We, however, quantify the impact of the KSplit marshaling protocol, and conduct an end-to-end performance measurement of an application using 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-replication invocation takes 502 cycles, which includes the overhead of executing the vmfunc instruction, saving and restoring general registers, and selecting 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 according to the active field’s type. 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
References


[23] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P:


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 recreating 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.