How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code

Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri

Abstract—In many modern operating systems (OSs), there exists no isolation between different kernel components, i.e., the failure of one component can affect the whole kernel. While microkernel OSs introduce address space separation for large parts of the OS, their improved fault isolation comes at the cost of performance. Despite significant improvements in modern microkernels, monolithic OSs like Linux are still prevalent in many systems. To achieve fault isolation in addition to high performance and code reuse in these systems, approaches to move only fractions of kernel code into user mode have been proposed. These approaches solely rely on static code analyses for deciding which code to isolate, neglecting dynamic properties like invocation frequencies.
We propose to augment static code analyses with runtime data to achieve better estimates of dynamic properties for common case operation. We assess the impact of runtime data on the decision what code to isolate and the impact of that decision on the performance of such "microkernelized" systems. We extend an existing tool chain to implement automated code partitioning for existing monolithic kernel code and validate our approach in a case study of two widely used Linux device drivers and a file system.

Index Terms—Operating Systems, Device Driver Isolation, Software Partitioning, Dynamic Program Analysis

1 Introduction

Modern operating system (OS) implementations either follow a monolithic or a microkernel architecture. Microkernel OSs strive to execute a bare minimum of their overall code base in privileged kernel mode [1]. Code that handles resource management, for instance, is separated in code that implements the actual resource (de-)allocation *mechanism* and code that implements the resource (de-)allocation *policy*. In microkernel OSs, only the former executes in kernel mode, which is sufficient to maintain non-interference of processes across shared resources. Monolithic OSs, on the contrary, execute a much larger fraction of their code base in kernel mode.

Traditionally, microkernel OSs were used for applications with high reliability requirements for two reasons. First, a small kernel code base is easier to understand and analyze, as the formal verification of the seL4 microkernel system demonstrates [2], [3]. Second, the effects of individual component failures at run time are limited to the respective components due to the fine-grained isolation among system components, which facilitates the implementation of sophisticated failover mechanisms (e.g., [4], [5], [6]).

Despite their reliability advantages over monolithic OSs, microkernels are seldom found in mobile, desktop, or server systems, even though reliability is a key concern for the latter. The reason for the dominance of monolithic systems in these areas lies in the poor IPC performance of early microkernel implementations, which led to significant overheads in operation. Although modern microkernels, such as the L4 family, feature highly optimized IPC implementations that reduce such overheads to a negligible fraction, their adoption is still mostly limited to embedded systems.

• *O. Schwahn, S. Winter, N. Coppik, and N. Suri are with the Department of Computer Science, Technische Universität Darmstadt, Robert-Piloty-Building, Hochschulstr. 10, 64289 Darmstadt, Germany.*
E-mail: {os, sw, nc, suri}@cs.tu-darmstadt.de

Ironically, the reason behind the predominance of monolithic OSs in commodity systems seems to be what is generally perceived as their major drawback. They execute large and complex code bases in privileged kernel mode within a single address space. For instance, Linux 4.7 comprised almost 22 million lines of code in July 2016 [7]. Reliable figures are difficult to obtain for proprietary OSs, but estimates for the Windows OS family are significantly higher [8].

On the one hand, this massive complexity entails severe threats to the reliability of OSs. As complex kernel code is difficult to develop and maintain, it is likely to contain software defects. Moreover, defects are likely to escape testing and other quality assurance measures since existing testing and verification techniques do not scale well to complex software. Such *residual defects* in kernel code have a high impact on system reliability if they get triggered during execution in privileged mode because there is no limit to the degree by which they can affect processes and system services. The risks of high software complexity have resulted in the proposal of small trusted/reliable computing base architectures (e.g., [2], [9], [10], [11]) for systems with high security or dependability requirements.

On the other hand, a large existing code base (and developer community that maintains it) also implies that the large amount of functionality it implements can be reused at low effort. Therefore, convenience has outweighed performance as a criterion for adopting a popular monolithic commodity OS over a microkernel OS. Early approaches like SawMill Linux [12] proposed to address this problem by manually integrating parts of Linux into the Exokernel and L4 OSs. Unfortunately, porting components across OSs is not a one-time effort and requires repeated manual adjustment as the forked code branches evolve. Ganapathy et al. developed an approach that addresses this problem by automatically splitting the kernel code of Linux at a per

function granularity into user and kernel mode portions [13]. The splitting is guided by a static set of rules that determine which code to allocate to which execution mode. While such automated splitting approaches cannot be expected to achieve all advantages of real microkernel OSs to the same degree, they still provide *better* isolation compared to a fully monolithic kernel without constraining code reuse. Unfortunately, the automated synchronization of data structures, locks, etc. between the user and kernel mode portions can entail performance overheads that exceed IPC induced overheads of microkernels by far.

Intuitively, these overheads highly depend on the kernel code *partitioning*, i.e., the decision which OS functions to execute in which mode. Moreover, the overheads depend on the system's application context. Function invocations across domains (from kernel to user mode or vice versa) entail performance overheads *per invocation*, making the performance of a partitioning dependent on dynamic system properties induced by the system's workload. The more frequent cross-domain invocations caused by a workload, the higher the overheads. Our paper addresses the central question *how to achieve a favorable partitioning that minimizes both performance overheads and the amount of code executing in kernel mode at the same time*.

Fig. 1 gives an overview of our proposed approach. We start from the source code of some kernel component (e.g., a driver) as input and produce a split mode version as output that is tailored to the user's application scenario. First, we extract static code properties, such as the static call graph, in a static analysis phase. We then generate an instrumented component version which is used to collect the dynamic usage profile under a typical workload. We then combine the statically and dynamically obtained data to formulate and solve the kernel component partitioning as 0-1 integer linear programming (ILP) problem. Our ILP formulation allows to fine-tune the trade-off between imposed overhead and amount of code that remains in the kernel. As final step, we synthesize a split mode version of the original component. The generated code for the split version is not intended for manual inspection or modification. Code maintenance, debugging, and evolution should still happen on the original code. Re-partitioning of evolved code is a simple mechanical task with our automated partitioning tool chain.

Our paper makes the following contributions:

- We propose to combine static and dynamic analyses to accurately assess the performance costs that moving kernel code to user mode would cause. Our assessment is automated and works on current Linux code.
- Using the dynamically recorded cost data, we model user/kernel mode partitioning of existing kernel code as 0-1 ILP problem and use the GNU Linear Programming Kit (GLPK) [14] to obtain a solution that achieves the desired trade-off between performance overhead and the size of the kernel mode code portion for improved isolation.
- We validate the utility of our approach by profiling and partitioning two device drivers and a file system in a case study and demonstrate the adaptability of the obtained partitioning to user requirements.

After a discussion of related work in Section 2, we introduce our system model in Section 3 and detail our profiling-based partitioning approach in Section 4. In Section 5, we demonstrate its utility by applying it to Linux kernel modules and compare the obtained partitionings and their performance characteristics. Section 6 summarizes insights gained from the results of our study and the required implementation work. Section 7 concludes the paper.

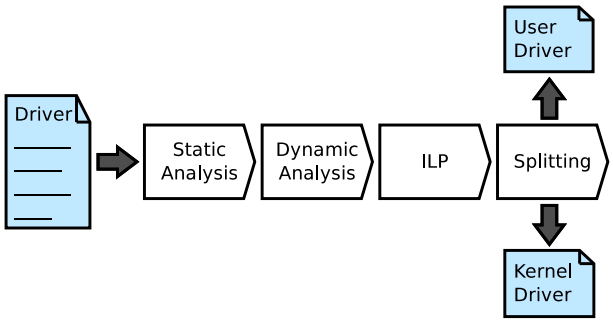

Fig. 1. Overview of the partitioning process, exemplified for a device driver. The input is the original driver source code and the output is a split mode version of the driver that implements the isolation/performance trade-off suitable for the user's application scenario.

2 Related Work

Software partitioning, also compartmentalization or disaggregation, is an important task in iterative software development and maintenance. Surprisingly, most research in this field has focused on the design of isolation mechanisms (i.e., *how* to isolate), whereas little work covers the actual partitioning process (i.e., *what* to isolate). Software partitioning has been proposed for a number of different isolation problems.

2.1 Privilege Separation

Privilege separation is a mechanism to prevent privilege escalation [15], i.e., the unauthorized acquisition of privileges through vulnerable programs. Privilege escalation vulnerabilities result from security-insensitive design that does not respect the principle of *least privilege* [16]. In execution environments that traditionally only support(ed) coarse grained execution privileges and access control, such as Unix and derivatives, implementing programs according to this principle has been challenging. As a consequence, a large body of legacy software does not employ newer, more fine-grained privilege separation mechanisms (e.g., [17]).

Privilege separation partitions programs into a *monitor* component, which executes privileged operations, and an unprivileged *slave* component such that vulnerabilities in the slave partition cannot lead to unauthorized execution of privileged operations. Although a large variety of mechanisms to isolate the slave from the monitor have been proposed in the literature [17], [18], [19], [20], [21], [22], the partitioning into suitable compartments is usually performed manually for few selected applications.

Privtrans [23] automates privilege separation for C programs based on user-supplied source code annotations that mark sensitive data and functions to be encapsulated by the monitor component. An automated data-flow analysis determines the monitor and slave partitions by propagating the annotations to all data and functions operating on or derived from annotated elements.

Wedge [24] extends the Linux kernel by several isolation primitives to implement privilege separation. To assist application partitioning into isolated compartments, the authors conduct dynamic binary instrumentation to derive interdependencies on shared memory between code blocks and their respective access modes from execution traces.

Jain et al. observe that capabilities in Linux are too coarse-grained to enforce the principle of least privilege for unprivileged users [25]. As a result, policies are commonly implemented in setuid-root binaries, a potential source of vulnerabilities. The authors present an extension of AppArmor which facilitates moving such policies to the kernel with minimal overhead.

Liu et al. employ combined static and dynamic analysis to automatically decompose an application into distinct compartments to protect user-defined sensitive data, such as private keys, from memory disclosure vulnerabilities [26].

2.2 Refactoring

Refactoring denotes the restructuring of software to improve non-functional properties without altering its functionality [27]. It comprises the decomposition of monolithic software systems as well as changes in an existing modular structure. Call graphs, module dependency graphs, or data dependency graphs are commonly used to represent software structures for refactoring (e.g., [28], [29], [30], [31]). Whether nodes in such graphs should be merged, split, or remain separate is usually decided by cohesion and coupling metrics [32] associated with the represented software structures, either by graph partitioning [28], [31], [33], cluster analysis [34], [35], [36], [37], or search based techniques [38], [39], [40].

2.3 Mobile/Cloud Partitioning

In order to enable sophisticated, computationally demanding applications to run on resource and energy constrained mobile devices, the partitioning of such applications into more and less demanding compartments has been proposed in the literature [41], [42], [43]. The former is then executed on the mobile device itself whereas the latter is executed remotely on a server infrastructure without draining the battery. Due to the dynamically changing operational conditions of mobile devices (battery strength, bandwidth, etc.), most approaches combine static and dynamic measures for partitioning, similar to the approach presented in this paper.

2.4 Fault Tolerance

A large number of approaches have been proposed to isolate critical kernel code from less critical kernel extensions, as existing extension mechanisms were found to threaten system stability in case of misbehaving extensions [44], [45], [46], [47], [48], [49]. Similar to privilege separation, most work in this field has focused on *how* to establish isolation between the kernel and its extensions [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], but only little work considers the problem of identifying *what* to isolate for achieving improved fault tolerance at an acceptable degree of performance degradation.

Ganapathy et al. target this question in the Microdrivers approach [13] that proposes a split-mode driver architecture, which supports the automated splitting of existing Linux device drivers into user and kernel compartments. The splitting is based on static analyses of the driver code and a set of static rules for classifying functions as either performance critical or uncritical. The approach has been implemented in a tool called "Driverslicer", a plugin for the CIL source code transformation and analysis tool chain [62], [63]. Renzelmann et al. extend Microdrivers to support Java for reimplementing the user part of split device drivers [64]. Butt et al. extend the Microdrivers architecture by security aspects using dynamically inferred likely data structure invariants to ensure kernel data structure integrity when data is transferred from the user part to the kernel part [65].

In our paper, we demonstrate that the addition of runtime information on performance measures significantly improves the partitioning by avoiding static worst-case approximations. We use this information to state partitioning as a 0-1 ILP problem, for which we obtain an *optimal solution* with respect to a given isolation/performance trade-off.

3 System Model

We consider the problem of bi-partitioning a kernel *software component* S (e.g., kernel modules) into a user mode fraction S^{υ} and a kernel mode fraction S^{κ} for split-mode operation to reduce kernel code size, where mode transitions occasion a cost c. We detail our software component model, cost model, and code size notion in the following subsections. This provides the foundations for stating kernel/user mode partitioning as 0-1 ILP problem in Section 4.

3.1 Software Component Model

As we target kernel code for partitioning, we assume S to be written in a procedural language like C. In procedural languages, a software component S comprises a finite set of *functions* $F(S) = \{f_i \mid i \in \mathbb{N}\}$[1]. Any function f_j can be *referenced* by any other function f_i of the same component and we denote such references by $f_i \rightsquigarrow f_j$. Our reference notion comprises direct (function calls) and indirect (passing function pointers as arguments) references [66]. Using the reference relation on functions, we obtain the *call graph* $(F(S), R(S))$, where $F(S)$ represent vertices and $R(S) = \{(a, b) \in F(S) \times F(S) \mid a \rightsquigarrow b\}$ edges of the graph.

3.1.1 Kernel Interactions

As allocating functions in S that heavily interact with kernel functions external to S to the user mode partition would significantly affect performance, we extend our software component model to describe such interactions. We have to consider two cases: (1) functions in S are invoked from other parts of the kernel not in S and (2) functions in S invoke kernel functions external to S. Hence, we add a *kernel node* $\mathfrak{K}$ and corresponding edges for references from and to such functions not in S to the call graph. We define the extended call graph as

$$\begin{aligned}(F'(S), R'(S)) = (F(S) \cup \{\mathfrak{K}\},\\ R(S) \cup \{(\mathfrak{K}, f) \mid f \in F_{entry}(S)\}\\ \cup \{(e, \mathfrak{K}) \mid e \in F_{ext}(S)\}),\end{aligned}$$

1. We do not include 0 in $\mathbb{N}$. In cases that include 0, we use $\mathbb{N}_0$.

where $F_{ext}(S) \subseteq F(S)$ is the set of functions that reference any function declared as `extern` in the program code of S, and $F_{entry}(S) \subseteq F(S)$ is the set of all functions on which the address-of operator (`&` in the C language) is used, i.e., functions potentially invoked by component-external code. Note that $\mathfrak{K}$ represents any function that resides within the kernel but is external to S, including core kernel functions as well as other in-kernel software components.

3.1.2 Data References

When loaded into memory, S resides in a memory *address space* $A(S) = [\bot_S, \top_S]$ with lower and upper bound addresses $\bot_S, \top_S \in \mathbb{N}_0$. S's data is contained in a finite amount of memory allocations $M(S) = \{(a, l) \mid a \in A(S) \wedge l \in \mathbb{N}\}$ of that address space, where a denotes the starting address of an allocation and l the *length* of the allocated slot in bytes. No memory allocation can exceed the address space boundaries:

$$\forall (a, l) \in M(S),\ a + l \leq \top_S$$

and memory allocations within an address space are disjoint:

$$\forall (a, l), (a', l') \in M(S),\ a < a' \Rightarrow a + l < a'.$$

We denote the reference (read/write access) of a function $f \in F'(S)$ to allocated memory $m \in M(S)$ by $f \leftrightarrows m$.

Note that interactions on shared memory are implicitly covered by our data model, as we do not require component address spaces to be disjoint. We assume that shared memory across differing address spaces is mapped to the same addresses in all address spaces and that memory allocation lengths are also the same for shared memory.

3.1.3 Partitioning

By bi-partitioning S's extended call graph $(F'(S), R'(S))$, we obtain two disjoint sets $F(S^\upsilon)$ and $F(S^\kappa)$ of functions, where functions $f \in F(S^\upsilon)$ reside in the user and functions $f \in F(S^\kappa)$ in the kernel mode partition. Note that the kernel node $\mathfrak{K}$ is, per definitions, always assigned to $F(S^\kappa)$. Moreover, we obtain three disjoint sets of edges:

$$R(S^\upsilon) = \{(a, b) \mid a, b \in F(S^\upsilon)\} \text{ and}$$
$$R(S^\kappa) = \{(a, b) \mid a, b \in F(S^\kappa)\}$$

are the sets of edges internal to the user and the kernel mode partitions, whereas

$$\begin{aligned} R_{cut}(S^\upsilon, S^\kappa) = \{(a, b) \in R'(S) \mid \\ (a \in F(S^\upsilon) \wedge b \in F(S^\kappa)) \\ \vee (a \in F(S^\kappa) \wedge b \in F(S^\upsilon))\} \end{aligned}$$

is the set of edges cut by the partitioning, i.e., edges that represent inter-domain function invocations. Neither nodes nor edges are lost during partitioning. So, we define the set of all possible partitionings of a software component S as

$$\begin{aligned} P_S = \{&(F(S^\upsilon), F(S^\kappa)) \mid \\ &F(S^\upsilon) \cap F(S^\kappa) = \emptyset \\ &\wedge F(S^\upsilon) \cup F(S^\kappa) = F'(S) \\ &\wedge R(S^\upsilon) \cup R(S^\kappa) \cup R_{cut}(S^\upsilon, S^\kappa) = R'(S)\}. \end{aligned} \tag{1}$$

The cost of the cut, and thereby the performance overhead of the partitioning, is then given by the sum of the weights of all edges in $R_{cut}(S^\upsilon, S^\kappa)$ and the isolation degree of a cut is expressed in terms of *size* of the S^κ partition; the smaller the kernel components the better the isolation. We detail both edge weights and size measures in the following.

3.2 Cost Model

To model the cost c associated with a partitioning $p \in P_S$ of a component S, we first define a weight function $w\colon R'(S) \to \mathbb{R}$ that assigns a weight to each edge of the extended call graph. The weight represents the expected overhead for invoking the corresponding reference as inter-domain function call. The associated overhead results from (a) mode switching overheads for changing the execution mode, (b) copying function parameters and return values between modes, and (c) synchronizing that part of the split component's state that is relevant to both partitions, i.e., memory locations m that are accessed from both partitions:

$$\begin{aligned} \{m \in M(S) \mid \exists f_\upsilon \in F(S^\upsilon), f_\kappa \in F(S^\kappa)\colon \\ f_\upsilon \leftrightarrows m \wedge f_\kappa \leftrightarrows m\}. \end{aligned}$$

Points (b) and (c) both require copying data between the disjoint memory allocations $M(S^\kappa)$ and $M(S^\upsilon)$ which imposes an overhead that depends on the amount of data to copy. The overall weight for each edge is therefore computed according to Eq. (2), where $t \in \mathbb{N}_0$ denotes the number of expected invocations of reference $r \in R'(S)$, $b\colon R'(S) \to \mathbb{R}$ denotes the average number of bytes transmitted upon a single invocation of a reference, and $c_{sys}\colon \mathbb{R} \to \mathbb{R}$ denotes the estimated time that mode switching and copying a number of bytes across partition boundaries takes on system sys. We detail the assessment of accurate edge weights using collected runtime data in Section 4.2.

$$w(r) = t \cdot c_{sys}(b(r)) \tag{2}$$

The cost for a partitioning $p \in P_S$ is given by $c : P_S \to \mathbb{R}$ as stated in Eq. (3), i.e., the sum of edge weights of all cut edges. By minimizing $c(p)$, we can find a partitioning with minimal cut weight, i.e., a partitioning with minimal overhead for inter-domain function calls.

$$c(p) = \sum_{r_i \in R_{cut}(S^\upsilon, S^\kappa)} w(r_i) \tag{3}$$

3.3 Isolation Degree

All software components S that execute in kernel mode do not only operate with the highest system privileges they also share the same address space, i.e., $\forall S_i, S_j,\ A(S_i) = A(S_j)$. Hence, defective or malicious code within such components could arbitrarily alter any code or data in any other kernel components and, ultimately, in the entire system. *Isolation* prevents the unintended alteration of a software component's data or code by another software component by enforcing domain boundaries between components that, if at all, can only be crossed via well defined interfaces. Intuitively, the degree of isolation in a system is higher the more code is executing in unprivileged user mode within a separate address space, as this code cannot directly access data or functionality in the kernel. We, therefore, measure the degree of isolation by the amount of kernel code executing in user mode, i.e., the user partition *size*.

To account for partition sizes, we assign all functions in the extended call graph their source lines of code (SLOC) count as node weight with $n\colon F'(S) \to \mathbb{N}_0$. The size of a

partition is then given by the sum of its node weights. As the kernel node $\mathfrak{K}$ represents the entirety of kernel functions external to S which, by definition, cannot be moved to the user mode partition, we define $n(\mathfrak{K}) = 0$ in order to include only component S in our size notion. Although the user partition size is a more intuitive measure for the isolation degree, we use the kernel partition size as a measure for *lack of isolation* in the following. The formulation of both optimization objectives, cut weight and partition size, as values to *minimize* facilitates their combination in a single cost function for optimization as we show later in Section 4.3. Due to the constraints on the node sets of user and kernel partition in Eq. (1), both size measures for isolation are equivalent.

We define $s\colon P_S \to \mathbb{N}_0$ accordingly in Eq. (4) for assigning a partitioning p its *lack of isolation* degree. A partitioning p with minimal $s(p)$ has the smallest possible amount of code residing in the kernel mode partition and, thus, the largest possible user mode partition, i.e., the highest isolation.

$$s(p) = \sum_{f_i \in F(S^{\kappa})} n(f_i) \tag{4}$$

4 Runtime Data Driven Partitioning

In order to obtain an ideal partitioning of a software component with respect to a desired isolation/performance trade-off according to our system model, we need to (1) perform a static code analysis to extract the component's call graph, the node weights (SLOCs), and the sets of possible data references from its program code, (2) perform a dynamic analysis of the component to assign edge weights (expected cross-mode invocation costs) to model the impact of partitioning on our objectives, and (3) formulate our optimization objectives and constraints as ILP problem.

To implement the approach outlined in Fig. 1, we reuse and extend the Microdrivers framework by Ganapathy, Renzelmann et al. [13], [64]. Originally, the framework only supported 32 bit (x86) Linux (v2.6.18), but we updated it to support contemporary 64 bit (x86_64) Linux versions (v3.14+). Our approach does not require modification of the Linux kernel beyond the component to be partitioned and, hence, is applicable to off-the-shelf kernels. Only the Microdrivers runtime and some parts of the tool chain may require updates for porting the approach to other kernel versions. We detail the individual processing steps in the following.

4.1 Static Analyses: Call Graph and Node Weights

We largely rely on the code analysis and transformation framework CIL [62], [63], which uses a simplified abstract representation of C code that can be analyzed by custom plugins written in OCaml. First, we use CIL to extract the static call graph from the input software component by identifying all defined functions and all function call sites. Second, we modify the obtained call graph according to our model and introduce the kernel node $\mathfrak{K}$ and corresponding edges. We handle indirect function invocations (via pointer) by adding edges to all functions whose signatures are compatible with the invoked function pointer type. This over-approximation introduces a number of false positives, i.e., edges that do not represent possible function calls during runtime. However, we compensate for these using the recorded runtime data from our dynamic analysis (cf. Section 4.2). Fig. 2 illustrates a resulting call graph, including node and edge weights.

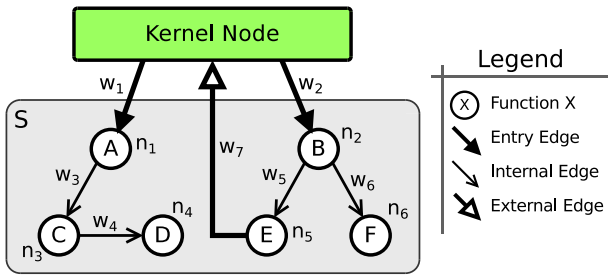

Fig. 2. Example call graph of a kernel software component S as used for partitioning. Nodes A to F represent functions with statically determined weights n_i; edges represent possible function calls with dynamically determined weights e_i. "Kernel Node" ($\mathfrak{K}$ in Section 3.1.1) represents all kernel functions outside component S.

For obtaining the node weights (n_i in Fig. 2), we analyze the software component's preprocessed C code and count the "physical" source lines of code (SLOC) for each function. We adopt the common SLOC notion and only include non-blank, non-comment lines. We implemented a Clang/LLVM based tool for extracting accurate SLOC counts on a per function level. We chose not to rely on CIL for this task in order not to distort the SLOC counts through CIL's code transformations, which generally increase the SLOC count disproportionately.

To extract the set of possible data references for each function, we reuse the marshaling (points-to) analysis of the Microdrivers framework, which is implemented as part of a CIL plugin called "Driverslicer". This is the same analysis that the Microdrivers framework employs for generating the marshaling code needed for synchronizing state between the user and kernel mode domains (see Section 3.2). The analysis yields an over-approximation of possible data references for each function, i.e., which data *may be reachable* from which functions. The analysis relies on programmer supplied code annotations as discussed later in Section 4.2.1. We refer the reader to the Microdrivers publications [13], [64] for a detailed discussion of Driverslicer's marshaling analysis. We use the results of this analysis in the dynamic analysis phase for collecting runtime data as detailed in the following section.

4.2 Dyn. Analyses: Edge Weights & Constrained Nodes

While static analyses are useful to obtain information related to the code structure, their utility to approximate function invocation frequencies or sizes of (dynamic) data structures is limited. For instance, invocation frequencies for function calls inside a loop that depends on input data can only be sensibly estimated by a dynamic analysis; the same is true for estimating the length/size of linked data structures such as lists or buffers whose size depends on input data. We compensate for this limitation by augmenting the statically obtained structure (call graph and node weights) with data from dynamic profiling. For edge weights, relying on recorded data from dynamic profiling yields more accurate results than static over-approximations, as long as the workload used to conduct the profiling is comparable to the system load encountered during actual operation.

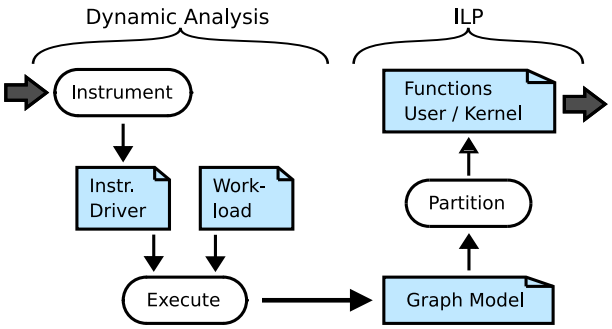

Fig. 3. Dynamic analysis and ILP steps in the partitioning process. An instrumented version of the original kernel module is built and executed under a given workload. The collected runtime profile is used to determine the edge weights in our graph model that is used for the ILP-based partitioning, which assigns all functions to either kernel or user mode.

4.2.1 Edge Instrumentation

For collecting the data needed to compute the edge weights (w_i in Fig. 2) according to our weight function $w(r)$ (cf. Eq. (2)), we instrument the software component S and execute it to capture its dynamic behavior under a given workload. A general overview of the dynamic analysis steps is given in the left part of Fig. 3.

We utilize the statically obtained call graph to identify relevant code locations for instrumentation. To collect data for all call graph edges that start in a node other than the kernel node $\mathfrak{K}$, i.e., edges $(f_i, f_j) \in R'(S),\ f_i \neq \mathfrak{K}$, we instrument all function call sites within S. For entry edges $(f_i, f_j) \in R'(S),\ f_i = \mathfrak{K} \wedge f_j \in F_{entry}(S)$, the call sites are external to S. Hence, we instrument the function bodies of the target functions f_j for these edges. For functions that can be invoked from within S as well as from $\mathfrak{K}$, we correct the collected entry edge data in a postprocessing step to avoid false accounting for entry edges.

We insert code at the above described code locations to record per edge: (i) the number of edge activations (function invocation frequency) (t in Eq. (2)), (ii) the estimated data amount that would be transmitted between functions in case of an inter-domain call as arguments and return values (an addend in the calculation of b in Eq. (2)), (iii) the estimated data amount for the synchronization of global data accessible from caller and callee (also contributing to b), and (iv) the *execution context* in which the call occurs. Information on the execution context is used to identify *constrained nodes*, i.e., nodes that cannot be moved to user mode, as discussed later in Section 4.2.2.

For the instrumentation, we employ aspect-oriented programming [67] techniques and generate the instrumentation code as separate C source code with our code generator tool, which is implemented as a CIL plugin. We use the *AspeCt-oriented C* compiler [68] to insert the instrumentation code into the component during the build process. Aspect-oriented programming has the advantage that the instrumentation code is written in the same language as the code that is being instrumented, while both can be maintained as separate modules.

The inserted code implements a dynamic size estimation by walking through data structures reachable from function parameters, global variables, and return values, and summing up their sizes. Linked data structures and heap allocated structures are handled correctly by following pointers and interpreting pointer targets according to the pointed-to data type. The required data type information for this estimation technique is obtained by reusing the points-to data from the Microdrivers marshaling analysis. The analysis relies on programmer supplied annotations to fill the gaps in the data type information inherent in the C language. For instance, annotations are required to resolve `void` pointers to actual types or to specify the length of dynamically allocated buffers. Effectively, we are refining the static data type based overestimation of reachable data structures that the Microdrivers analysis provides using actual data values observed during runtime. For instance, if we observe a `NULL` pointer in a data structure, we do not consider the pointer target's data type for size estimation. The described approach is tailored for use with the Microdrivers framework. If another framework is selected to implement the split mode operation, the size estimation has to be adapted to reflect the data synchronization approach of that framework.

Using the recorded invocation frequencies and data transmission estimates from the dynamic analysis, we can derive the expected performance overhead that cutting edges in $R'(S)$ implies. As such overhead differs on different hardware platforms (and also with different frameworks used for splitting), we express the actual cost as a function c_{sys} of the amount of data to be copied. To determine c_{sys}, we implemented a kernel module for conducting measurements on the target platform, where the split mode component should ultimately execute. The module measures and records the overhead that the transfer of different data amounts causes in inter-domain function invocations. We fit a function onto the recorded data and use it to estimate the overhead for the average data sizes recorded during profiling. This completes the information required for calculating edge weights according to Eq. (2): t is the number of observed edge activations, $c_{sys}(x)$ is the fitted platform dependent function, and $b(r)$ is the average number of bytes transmitted.

4.2.2 Constrained nodes

Due to the structure of commodity OSs, and in particular Linux, there are functions (nodes) that have to remain in the kernel partition. The auxiliary node $\mathfrak{K}$, representing all functions external to S, must remain in the kernel partition by definition. Another example are functions that may execute in interrupt context. This is an inherent limitation of the Microdrivers framework, which synchronizes between user and kernel mode via blocking functions and code running in interrupt context cannot sleep [69]. Consequently, we must ensure that such non-movable functions remain in the kernel partition. A number of possibilities exist to circumvent this restriction, for instance by changing the synchronization mechanisms in Microdrivers or by employing mechanisms for user mode interrupt handling, such as in VFIO [70] or the Real-Time Linux patch set [71]. As these only affect the achievable partitioning result and not the partitioning approach, which is the central topic of this paper, we do not assess the impact of these options.

We denote $F_{mov}(S) \subseteq F'(S)$ as the set of movable functions and $F_{fix}(S) \subseteq F'(S)$ as the set of functions that are fixed in kernel mode. Both sets are disjoint and $F_{mov}(S) \cup F_{fix}(S) = F'(S)$. We determine $F_{fix}(S)$ using the

execution context records from profiling. Every function f_i that executed in interrupt context during profiling and all functions f_j that are reachable from f_i are in $F_{fix}(S)$ (transitive closure). Note that this approach may miss some unmovable functions if they were not observed in interrupt context. Such false negatives can be mitigated in the resulting partitioning, for instance, by providing alternate code paths that allow the execution of interrupt functions within the kernel even though they were moved into the user mode partition. However, we did not encounter any such cases in our case study.

A number of kernel library functions, e.g., string functions like `strlen`, have equivalent functions in user mode libraries. These functions can be ignored for the partitioning as a version of them exists in both domains. We therefore remove them from our call graph model prior to partitioning. This is a performance optimization for the resulting split mode components.

4.3 Partitioning as 0-1 ILP Problem

We express our partitioning problem as 0-1 ILP problem, as illustrated in the right half in Fig. 3. In general, stating a 0-1 ILP problem requires a set of boolean *decision variables*, a linear *objective function* on the variables to minimize or maximize, and a set of linear inequalities as *problem constraints* over the variables. Once stated as ILP problem, a linear solver can be used to find an *optimal* partitioning.

4.3.1 Decision Variables

We introduce the following two sets of boolean variables: $x_i, y_i \in \{0,1\}$. For each node f_i in our call graph, a corresponding variable x_i assigns f_i to either the user or kernel mode partition as follows:

$$\forall f_i \in F'(S),\ x_i = 0 \Leftrightarrow f_i \in F(S^{\upsilon}) \wedge x_i = 1 \Leftrightarrow f_i \in F(S^{\kappa})$$

Additionally, a variable y_i determines for each corresponding call graph edge r_i whether the edge is cut by the partitioning as follows:

$$\forall r_i \in R'(S),\ y_i = 0 \Leftrightarrow r_i \notin R_{cut} \wedge y_i = 1 \Leftrightarrow r_i \in R_{cut}.$$

4.3.2 Problem Constraints

Since variables x_i and y_i are boolean, we can express their relation using a boolean exclusive-or (XOR) operation $y_k = x_i \oplus x_j$, where y_k encodes if edges $r_k = (f_i, f_j)$ are cut or not and x_i, x_j represent the partition assignments of the two adjacent nodes. In order to express this relation as a linear equation system, we define four constrains for each edge as given in Eqs. (5) to (8). The constraints encode the boolean truth table for XOR, one equation per row in the truth table.

$$x_i + x_j - y_k \geq 0 \quad (5)$$
$$x_i - x_j - y_k \leq 0 \quad (6)$$
$$x_j - x_i - y_k \leq 0 \quad (7)$$
$$x_i + x_j + y_k \leq 2 \quad (8)$$

In addition to the XOR encoding, we need further constrains to fix non-movable functions as discussed above in the kernel partition, i.e., $\forall f_i \in F_{fix}(S),\ x_i = 0$. We achieve this by adding one additional constraint of the form given in Eq. (9) per non-movable function f_i.

$$x_i \leq 0 \quad (9)$$

4.3.3 Objective Function

We combine the cost (Eq. (3)) and size (Eq. (4)) functions from Section 3 to a single objective function with a balance parameter $\lambda \in [0,1]$. We compute the edge weights w_i and node weights n_i as described above for all functions and edges in $(F'(S), R'(S))$. We then reformulate our minimization objectives $c(p)$ and $s(p)$ as sums over normalized edge and node weights including decision variables as defined in Eqs. (10) and (11). The node and edge weights are normalized to the interval $[0,1]$ according to Eq. (12) which also normalizes both equations. The normalized weights represent percentages of the overall weight present in the call graph.

$$c'(S) = \sum_{r_i \in R'(S)} ||w_i|| \cdot y_i \quad (10)$$

$$s'(S) = \sum_{f_i \in F'(S)} ||n_i|| \cdot x_i \quad (11)$$

$$||a_i|| = \frac{a_i}{\sum_{j=1}^{n} a_j} \quad (12)$$

Combining Eqs. (10) and (11) into one linear function with a balance parameter λ yields Eq. (13), which is our final objective function for the ILP solver.

$$obj(S) = \lambda \cdot c'(S) + (1-\lambda) \cdot s'(S) \quad (13)$$

λ allows to tune the trade-off between the expected performance overhead and the amount of code that resides in the user partition. Setting λ to a value near 1 prioritizes the minimization of the performance overhead, i.e., cut cost $c(p)$. In this case, a resulting partitioning can be expected to have a near zero cut cost, i.e., negligible performance overhead, but a large kernel partition. Setting λ to a value near 0 prioritizes the minimization of the kernel partition, i.e., SLOC count $s(p)$. A partitioning in this case can be expected to have a kernel partition as small as possible, but a high performance overhead.

5 Evaluation

We demonstrate the utility of our approach in a case study of three Linux kernel modules: two device drivers (`psmouse` and `8139too`) and one file system (`romfs`). For the dynamic analyses, we expose the instrumented kernel modules to throughput benchmarks and collect their runtime profiles. We derive the platform overhead functions c_{sys} for our target systems from additional measurements and use them with the obtained profiles for generating and comparing partitionings with different isolation/performance trade-offs. We highlight general insights from this process that are not limited to the scope of our case study.

5.1 Experimental Setup

We use two test machine setups for our evaluation: (1) a physical machine setup (PHY) and (2) a virtual machine setup (VM). Both systems run Debian 8.4 (Jessy) with Linux 3.14.65 (long-term support) in a 64-bit (x86_64) configuration.

TABLE 1
Overview of the selected test modules. SLOC columns report original and preprocessed line numbers. Function columns list the number of overall and entry functions. The remaining columns list the number of external & library functions and call sites.

	SLOC		Functions				
Module	Orig	PreProc	All	Entry	Extern	Lib	Calls
8139too	2087	38 042	123	35	69	19	378
psmouse	1390	20 779	59	26	42	2	214
romfs	927	27 448	42	14	25	4	96

Our physical machine is equipped with a quad-core Intel i7-4790 CPU @ 3.6 GHz, 16 GiB of RAM, a 500 GB SSD, and a 200 GB HDD. Our virtual system emulates a dual-core CPU and 1 GiB of RAM. As virtualization platform, we employ QEMU/KVM 2.1.2 using the just described physical machine as host. Note that we use the VM setup only for `psmouse` experiments as we rely on QEMU's ability to emulate mouse events. The HDD is used for `romfs` experiments.

5.1.1 Test Module Selection

To demonstrate the applicability of our approach to general in-kernel components, we select kernel modules that utilize distinct kernel interfaces and exhibit different runtime characteristics for our evaluation. Table 1 lists the kernel modules we selected for that purpose. `8139too` is the driver for RealTek RTL-8139 Fast Ethernet NICs, which executes mostly in interrupt context and interacts with the kernel's networking subsystem. `psmouse` is the driver for most serial pointing devices (mouse, trackpads), which executes largely in interrupt context, has complex device detection and configuration logic, and interacts with the kernel's serial I/O and input subsystems. `romfs` is a read-only file system used for embedded systems; it does not execute in interrupt context and interacts with the kernel's virtual file system and block I/O infrastructure.

Table 1 shows static size metrics for the selected test modules. The *SLOC* columns list the physical source lines of code[2] before and after code preprocessing (as part of the build process). The *Functions* columns list the number $|F(S)|$ of functions implemented in the module (*All*) and the number $|F_{entry}(S)|$ of entry point functions (*Entry*). Column *Extern* lists the number of external functions referenced, *Lib* lists the number of library functions that exist in both kernel and user mode, and *Calls* lists the number $|R'(S)|$ of calls (references). Judging by the presented numbers, `8139too` is the most complex of the modules with more code, more functions, and a larger interface with the kernel, which results in a higher coupling with other kernel subsystems than the other modules. The relatively high number of entry functions for `psmouse` is due the driver's heavy usage of function pointers rather than a large exported interface (see Section 3.1.1).

5.1.2 Workload Selection

As workloads, we apply throughput benchmarks with a duration of 60 s to all test modules. For `8139too`, we use `netperf` 2.6 in *TCP_STREAM* mode measuring the network throughput. For `psmouse`, we use QEMU's monitor and control interface (QMP) to generate mouse move events measuring the event throughput. For `romfs`, we use `fio` 2.2.9 to perform file read tests measuring read throughput. All workloads contain the module loading/unloading steps and all initialization/cleanup operations, such as `mount`/`umount` for `romfs` and `ifup`/`ifdown` for `8139too`.

2. generated using David A. Wheeler's SLOCCount

TABLE 2
Runtime profile overview. The first four columns list the number of activated function nodes and call edges (absolute and relative); the last column reports the relative amount of movable functions.

	Activations		Rel. Coverage		
S	$F(S)$	$R'(S)$	$F(S)$	$R'(S)$	$F_{mov}(S)/F(S)$
8139too	82	201	66.7 %	53.2 %	65.9 %
psmouse	36	81	61.0 %	37.9 %	83.1 %
romfs	36	84	85.7 %	87.5 %	83.3 %

5.2 Instrumentation & Profiling

We instrument all modules with our aspect-oriented instrumentation tool and execute them in our test systems using the aforementioned workloads. The instrumented modules are only used to collect profiling data; they are removed from the system once profiling is complete. We repeat the profiling runs 50 times for each module, rebooting the systems before each run to avoid interferences between runs. In addition to profiling runs, we also perform 50 runs with the non-instrumented module as a baseline to determine the runtime overhead incurred by the instrumentation and the split mode operation.

5.2.1 Instrumentation Overhead

In terms of binary module size, the instrumented module versions are about 12 to 42 times larger than the original ones. This is due to the aspect-oriented instrumentation approach, which produces additional C code for each function call site. We report performance measurements for the instrumented and the original module versions in Table 3 together with our overall results. As apparent from columns *Throughput* and *Init/Clean Ops*, the instrumentation does not impact throughputs or init/cleanup times. Module load/unload times increase slightly for some modules, with a maximum increase of factor 3.6 for loading `8139too`. We therefore conclude:

> *Aspect-orientation provides a modular way to implement source code instrumentation on the abstraction level of the targeted programming language with overheads small enough to allow production usage.*

5.2.2 Runtime Profiles

Table 2 gives an overview of the observed runtime profiles. The *Activations* columns list the number of functions and references that our workload activated, whereas the *Rel. Coverage* columns report the relative amount of activations. The last column reports the percentage of functions that our partitioner may move to the user mode partition, i.e., the number $|F_{mov}(S)|$ of nodes for which no constraints apply (cf. Section 4.2.1). Our `romfs` workload achieves the highest coverage as this module only contains the essentials for reading from the file system. The percentage of constrained

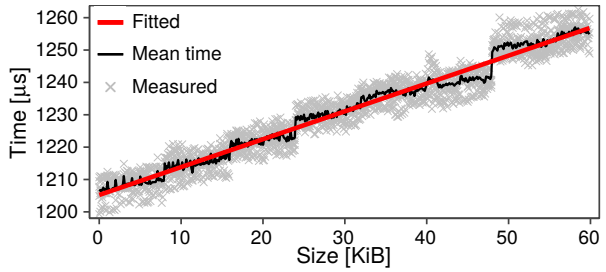

Fig. 4. Platform overhead c_{sys} for data sizes from 0 to 60 KiB measured for our physical machine setup.

nodes is lowest here as `romfs` does not execute in interrupt context and only needs a few functions fixed in the kernel to ensure correct operation in split mode. The relatively low percentage of activated calls in `psmouse` is due to the usage of function pointers as well as the high amount of device specific detection and configuration logic, most of which is not needed for our emulated QEMU mouse device. The few unmovable functions in this module execute in interrupt context. `8139too` has the lowest fraction of movable functions as this driver primarily executes in interrupt context for network package handling. In summary, there is significant potential for moving functions to user mode for `psmouse` and `romfs` since only a small fraction of functions needs to be fixed in the kernel. The potential for moving many functions without severe performance implications is particularly high for `psmouse` and `8139too` in the given usage scenario as functions without activations can be moved to user mode without affecting the performance under the respective common case usage.

5.3 Estimation of the Platform Overhead

We estimate the platform overhead function c_{sys} for both our setups (PHY & VM) using a split mode test module (cf. Section 4.2) that measures the time needed for inter-domain function invocations with data of increasing sizes up to 60 KiB in 128 B steps. All data sizes recorded during profiling fall into this range. For each size step, we measure 1000 inter-domain calls and use their average time as the result for each step. We repeat the overall measurement process 10 times and fit a linear function onto the average measurements as we are interested in getting a mean overhead estimation.

Fig. 4 illustrates the results for our PHY system. The horizontal axis shows the size in KiB whereas the vertical axis shows the measured time in microseconds. We omit the plot for the VM setup as it is very similar with a slightly larger vertical offset. The fitted linear function for our PHY setup can be written as $c_{sys}(b) = 1205.2 + 0.00084 \cdot b$ (coefficient of determination $r^2 = 0.98$), and the function for our VM setup as $c_{sys}(b) = 1259.7 + 0.00082 \cdot b$ (with $r^2 = 0.97$). For both systems, there is a considerable static overhead of about 1205 µs for PHY and about 1260 µs for VM associated with every inter-domain function invocation. The actual data transfer entails a much smaller overhead of about 0.8 µs per 1 KiB. In earlier experiments on Linux 3.14.40, we observed a static overhead of 1214 µs for PHY, indicating that c_{sys} results may be reused across different revisions of the same kernel.

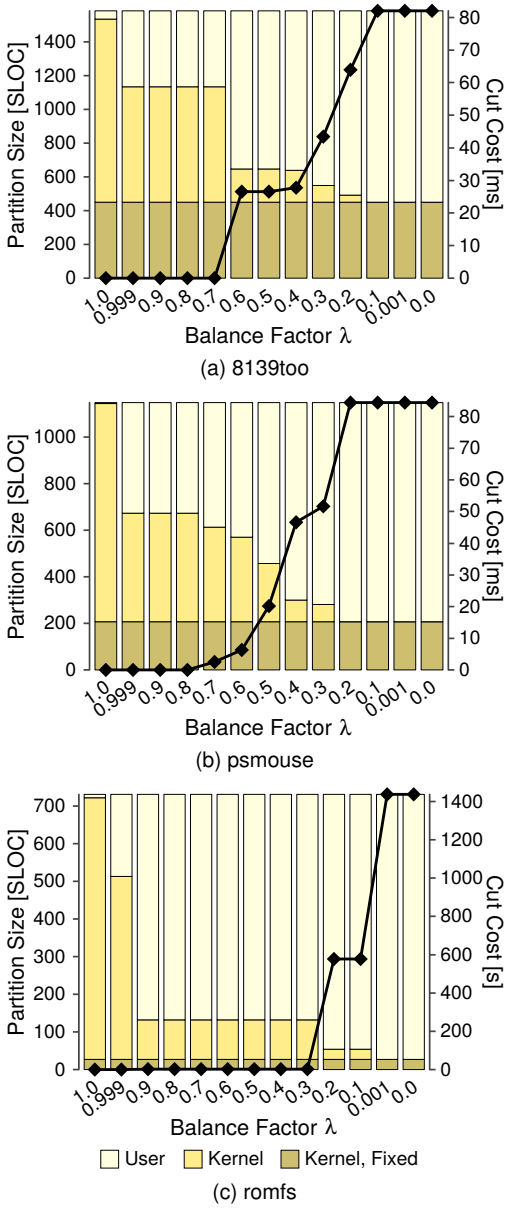

Fig. 5. Development of partition sizes (left axis) and cut costs (right axis) for varying λ values (decreasing left to right) for our three test modules. The kernel partition size decreases with decreasing λ whereas cut costs increase, i.e., the more code is moved to the user partition, the higher the performance impact.

5.4 Partitioning Results

We use the GLPK IP/MIP solver v4.55 [14] for partitioning. We generate 13 partitionings per module using different λ values to investigate the effect on the resulting partitions. For `8139too`, the solver needs on average about 1.3 MiB of RAM with 362 decision variables (after problem preprocessing). For `psmouse`, it uses about 0.8 MiB with 241 decision variables; `romfs` needs 0.4 MiB with 128 decision variables. The solver runtimes are negligible as they are reported with 0.0 s in all runs. These numbers demonstrate that, although 0-1 ILP problems are generally NP-complete, our optimization-based partitioning approach is suitable for realistic problem sizes.

If stated as 0-1 ILP problem, optimal partitioning of real-world software components can be achieved with modest computational overhead.

Figs. 5a to 5c illustrate the sizes ($s(p)$) and cut costs ($c(p)$) of the generated partitions. The horizontal axes display the used λ values (being identical for all modules) whereas the left vertical axes show the sizes for kernel and user partitions (in SLOC); the right axes show the cut costs (in time units). Note that the figures use different scales on the vertical axes, with Fig. 5c using seconds and the others using milliseconds.

Obviously, the amount of fixed kernel code does not change with varying λ, i.e., the minimal kernel partition size is bounded by the amount of non-movable (interrupt) functions. Nonetheless, the overall kernel partition sizes decrease with decreasing λ as the minimization of $s(p)$ gains priority. The cost, however, increases with decreasing kernel partition size as more and heavier graph edges are cut with less priority on the minimization of $c(p)$. As λ approaches 0.0, a high cost must be payed even for small decreases in the kernel partition size.

If interrupt handling in device drivers was revised to allow for execution in process context, larger portions of their code could be isolated as user mode processes.

For all modules, the kernel partition is smallest at $\lambda = 0$ with highest cut cost. For $\lambda = 1$, the opposite is the case. Decreasing λ from 1.0 to 0.999 allows the solver to find a partitioning that not only has a low cut cost, but also a larger user mode partition. This effect occurs as the solver solely minimizes the cut cost at $\lambda = 1$ without taking any node weights into account, i.e., *any* partitioning having minimal cost (in our scenario 0) is optimal for the solver, irrespective of the SLOC counts left in either partition. This is also the reason why even with $\lambda = 1$, we still have some code fractions left in the user partition. Putting a little effort into node weight minimization, however, is enough for the solver to move all "cheap" nodes. In other words, all nodes that the solver can move "for free" under a given workload are actually moved to the user partition. This gives the benefit of a higher isolation, while a performance overhead must be paid only in rare occasions that are outside the common case usage. Therefore, we recommend to not select $\lambda = 1$ but close to 1. A similar effect does not occur when we increase λ from 0.0 to 0.001 as there is no way to reduce cut costs without increasing the kernel partition size. Note that not all generated partitions are distinct as different (neighboring) λ values may result in the same partitioning. This is due to the node and edge weights being discrete (a function can only be moved as a whole). Our partitioner produces 7 distinct partitionings for `8139too`, 8 for `psmouse`, and 5 for `romfs`. Table 3 reports all distinct partitionings in the *Partition* columns.

For a known usage profile, significant portions of kernel software components can be isolated at near zero performance overhead in the common case.

Although `romfs` appears to be the simplest module from the static metrics presented in Table 1, we expect especially large overheads as the cut costs illustrated in Fig. 5c are very high compared to the other two test modules. This is due to the nature of `romfs`, which moves large data chunks with high call frequencies between disk and memory. Even the partitioning with $\lambda = 0.9$ already has a cut cost of about $2.1\,\mathrm{s}$. This effect is due to the edge weight normalization (see Eq. (12)), which is applied to formulate the overall minimization problem for the solver. Workloads that lead to extreme hot spots in terms of call frequencies and/or data amount in the runtime profile require finer grained λ variations around 1.0 if the maximum size user partition with zero cost should be found, since the hot spots in the profile dominate the partitioning cost.

Information on a software component's dynamic usage profile is essential for an accurate cost estimation.

After automatically generating a spectrum of partitionings with different λ values using our approach, a system administrator can select a partitioning with the performance/isolation trade-off that best fits the requirements of the intended application scenario. Choosing the lowest λ value that meets required execution latencies, for instance, yields best effort reliability.

5.5 Split Mode Modules

We synthesize and build split mode modules for all distinct partitionings that we generated and expose them to our workloads for timing and throughput measurements. Table 3 reports the results. We highlight especially interesting numbers in bold face.

Overall, split modules with a cut cost of zero do not show different performance compared to the original modules except for slightly increased loading times caused by the initialization of the Microdrivers runtime. The measured throughputs for the two interrupt heavy drivers (`8139too` and `psmouse`) remain stable as interrupt routines are not touched due to earlier discussed Microdrivers limitations. As soon as the estimated cut costs increase beyond zero, we observe a modest impact on operation latencies for `8139too` and `romfs`. Load times increase for both modules as well as `mount` and `unload` times for `romfs` and `ifup` time for `8139too`. The observed increases are due to the assignment of module and device initialization/configuration functions to user mode.

For `psmouse`, an increase in times becomes apparent only in later partitionings: starting with $\lambda = 0.4$, the module load times increase to $1.5\,\mathrm{s}$ as all the device detection and initialization logic gets moved to user mode. Although our estimated costs also make a jump for this partitioning, it is far smaller than the measured overhead. We attribute this anomaly to side effects that our model does not account for. `psmouse` initialization logic causes additional interrupts that interfere with the user mode process executing the mouse logic, which leads to more context switches and wait times for the user mode process.

All modules exhibit the largest performance decrease when the cut cost is highest and the user mode partition is largest. The measured time and performance impacts for `8139too` and `psmouse` are not prohibitively high for use in production. This is consistent with the estimated cut costs that remain below $100\,\mathrm{ms}$. In contrast, `romfs` suffers from a significant decrease of two orders of magnitude in

TABLE 3
Partitioning results and performance measurements for `8139too`, `psmouse`, and `romfs`. Each row reports average results of 50 experiment runs (standard deviation in brackets). Partition columns report size and estimated cost of the respective partitioning. The other columns report times for module loading/unloading and initialization/cleanup operations. The last column lists workload throughput. All times are reported in milliseconds, except for partitioning costs of `romfs`, which are in seconds. Interesting data points are highlighted in bold.

Version	Partition			Module Op (ms)		Init/Clean Ops (ms)		Throughput
	Kern	User	Cost	load	unload			
			8139too			ifup	ifdown	Mbit/s
orig	-	-	-	5.0 (3.0)	45.6 (12.4)	44.5 (1.5)	7.6 (0.1)	94.1 (0.0)
instrumented	-	-	-	17.8 (4.9)	44.2 (12.9)	44.6 (1.6)	7.6 (0.1)	94.1 (0.1)
split, $\lambda = 1.0$	1535	49	0.0	9.9 (0.1)	45.8 (13.7)	44.5 (1.4)	7.6 (0.1)	94.1 (0.0)
split, $\lambda = 0.7$	1134	450	0.0	14.0 (0.1)	48.4 (13.7)	44.8 (1.5)	7.6 (0.1)	94.2 (0.0)
split, $\lambda = 0.5$	647	937	26.6	**34.8 (0.3)**	47.8 (12.5)	**52.2 (1.4)**	7.5 (0.1)	94.1 (0.0)
split, $\lambda = 0.4$	639	945	27.8	34.8 (0.3)	42.8 (11.6)	52.4 (1.2)	7.6 (0.1)	94.2 (0.0)
split, $\lambda = 0.3$	549	1035	43.4	35.0 (0.2)	47.1 (12.2)	59.7 (1.2)	18.8 (0.2)	94.1 (0.0)
split, $\lambda = 0.2$	492	1092	63.9	39.2 (0.3)	72.7 (12.0)	59.3 (1.5)	18.7 (0.2)	94.2 (0.0)
split, $\lambda = 0.0$	450	1134	82.0	**46.8 (1.4)**	**73.5 (14.2)**	**64.2 (1.4)**	**23.3 (0.2)**	94.1 (0.2)
			psmouse					Ev/s
orig	-	-	-	3.5 (1.5)	57.8 (23.1)			1060.3 (13.0)
instrumented	-	-	-	4.4 (1.3)	67.1 (19.4)			1064.2 (16.7)
split, $\lambda = 1.0$	1144	4	0.0	5.6 (2.4)	62.6 (21.6)			1059.8 (9.7)
split, $\lambda = 0.8$	673	475	0.0	5.7 (1.8)	60.8 (24.4)			1057.7 (11.1)
split, $\lambda = 0.7$	613	535	2.5	5.5 (2.4)	63.6 (21.4)			1060.6 (9.4)
split, $\lambda = 0.6$	570	578	6.3	5.1 (1.6)	59.5 (23.5)			1056.6 (7.9)
split, $\lambda = 0.5$	457	691	20.2	5.4 (1.7)	62.8 (24.9)			1056.1 (10.4)
split, $\lambda = 0.4$	300	848	46.6	**1545.3 (22.6)**	63.8 (22.1)			1057.1 (10.7)
split, $\lambda = 0.3$	281	867	51.7	**1540.0 (19.3)**	65.1 (22.4)			1059.9 (7.8)
split, $\lambda = 0.0$	207	941	84.4	**1537.8 (17.1)**	**114.5 (30.4)**			1060.1 (8.5)
			romfs			mount	umount	MiB/s
orig	-	-	-	2.8 (1.9)	39.6 (10.2)	1.1 (1.4)	101.7 (8.1)	33.76 (0.72)
instrumented	-	-	-	5.7 (3.3)	36.9 (8.3)	1.1 (1.2)	100.3 (9.0)	33.62 (0.59)
split, $\lambda = 1.0$	722	9	0.0	5.3 (0.1)	35.7 (11.2)	1.2 (1.6)	99.4 (9.3)	33.60 (0.53)
split, $\lambda = 0.999$	513	218	0.024	**10.5 (0.2)**	**44.2 (11.1)**	**22.4 (1.1)**	103.8 (11.8)	33.69 (0.55)
split, $\lambda = 0.3$	132	599	2.1	11.4 (0.3)	44.0 (10.2)	37.7 (1.5)	**187.1 (11.2)**	33.73 (0.62)
split, $\lambda = 0.1$	54	677	577.5	11.9 (0.2)	45.2 (9.2)	37.6 (1.2)	**126.3 (9.5)**	**0.288 (0.00)**
split, $\lambda = 0.0$	27	704	1437.4	11.9 (0.3)	**46.3 (12.1)**	**41.0 (0.8)**	**119.4 (9.3)**	**0.065 (0.00)**

throughput starting from $\lambda = 0.1$ as the function that transfers contents between disk and memory (`romfs_readpage`) is moved to user mode. This is expected as the estimated cut cost becomes exceptionally high for large user mode partitions with about 24 min for $\lambda = 0.0$. The decrease in `umount` times between splits with $\lambda = 0.3$ and $\lambda = 0.1$ is a side effect of the observed throughput decrease. During unmounting, `romfs` cleans up per-file i-node data structures using a function that is moved to user mode starting at $\lambda = 0.3$. Due to the lower throughput, fewer files are read as part of our fixed duration workload. Hence, fewer i-nodes need to be cleaned up and `umount` needs less time.

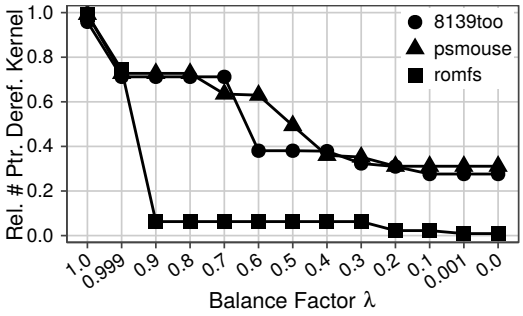

Fig. 6. Relative amount of pointer dereferences in the kernel mode partition across different λ values. With decreasing λ, the size of the kernel mode partition decreases (left to right) along with the amount of potentially dangerous pointer dereferences.

5.6 Reliability of Split Mode Modules

To assess the reliability gain of split mode modules, we conduct both a code analysis and fault injection experiments. As memory safety bugs constitute an important class of program bugs in C code, we focus on potentially invalid memory accesses via pointers. We analyzed the source code of our test modules to identify all code locations where pointers are dereferenced. In case of corrupted pointer values, dereferences can lead to invalid memory accesses, which, depending on the accessed memory location and whether it is a read or write access, can crash the kernel. Fig. 6 illustrates the relative amount of pointer dereferences that are left in the kernel mode partition with decreasing kernel mode size (decreasing λ). The smaller the kernel partition, the fewer potentially dangerous pointer dereferences are left inside the kernel.

Overall, we found 507 dereferences in `8139too`, 389 in `psmouse`, and 223 in `romfs`. For the smallest kernel mode partition, the amount of dereferences falls below 31 % for `8139too` and `psmouse` and for `romfs` even below 1 %. The dereferences remaining in the kernel partition reside in non-movable functions. However, at least 50 % of dereferences can be removed from the kernel partition at moderate cost (see $\lambda = 0.5$) for all three modules.

To validate that invalid memory accesses are indeed a problem when they occur inside the kernel but can be tolerated in the user partition, we conduct fault injection experiments in which we inject `NULL` pointer values into the previously identified pointer dereferences via code mutation, a fault type that kernel code is particularly prone to [49]. We randomly selected 50 mutants per module, compiled them and executed them with our workload for all previously generated partitionings. With a total of 20 distinct splits across all mutants, this sums up to a total of 1000 experiments. Across all experiments, 46 % of injected faults got activated for `romfs`, 78 % for `psmouse`, and 70 % for `8139too`. In all experiments with activated faults, the kernel reacted with an Oops and required rebooting if the invalid memory access resided in the kernel partition. However, if the invalid access resided in the user partition, the user mode driver process reacted with a segmentation fault, leaving the rest of the system unaffected. When the kernel partition size was largest ($\lambda = 1.0$), we observed an Oops in 100 % of cases. However, if the kernel partition size was smallest ($\lambda = 0.0$), we observed Oopses only in 44 % of cases for `psmouse`, 6 % for `8139too`, and 9 % `romfs`. We conclude that the more code we move into the user partition the more potential invalid memory accesses can be isolated in the user mode driver process, thereby improving system reliability.

6 Discussion

Our evaluation demonstrates that kernel components can be partitioned into user and kernel compartments based on data recorded from runtime profiling while allowing for a user-defined trade-off between isolation and runtime overhead. For `romfs`, the cost for a minimal kernel component is prohibitively high, but other λ values yield usable partitionings with overheads corresponding to the amount of kernel functionality isolated. In the following, we summarize issues, insights, and practical considerations from our implementation.

A basic assumption of the used Microdrivers framework is that *data accesses are properly synchronized in the original kernel module using locking primitives*, so that shared state in the split module needs to be synchronized only upon the start and end of inter-domain function invocations and whenever a locking operation is performed. In reality, however, locks are often avoided for performance reasons, especially for small, frequently updated data fields such as the `flags` field in the `page` struct, which the Linux kernel uses for page management. Here, atomic access operations are commonly used. Atomic operations need special handling in split mode modules as the accessed data must be synchronized immediately upon access. For this reason, we left all accesses to certain fields of the `page` struct in the kernel. For instance, the `romfs_readpage` function only started working in user mode after we ensured that all page status bit accesses were left in the kernel, which reads/writes these fields concurrently using CPU-specific atomic memory operations.

State synchronization in the presence of interrupts has also proven challenging. Especially during device initialization and configuration, drivers issue device commands that may result in immediate interrupts, i.e., the driver code interrupts itself. As these commands are not automatically identified, the Microdrivers runtime is unaware of the resulting control flow redirection to the interrupt service routine. Therefore, no data synchronization is performed before the interrupt-causing operation is executed and, hence, the interrupt routine and user mode function may operate on inconsistent copies of shared data. We encountered this issue with both `8139too` and `psmouse`. Device configuration from user mode only worked after we added additional synchronization points and left certain operations in the kernel.

While we do not consider security as a partitioning goal, we note that the presented approach does not harm security: In a properly configured system, the interface between the kernel and user components of a split mode driver is not accessible to unprivileged users. Consequently, the attack surface remains the same as for the original driver. Moreover, moving vulnerable code from the kernel to user space can reduce the severity of vulnerabilities. The user component can also benefit from hardening and mitigation techniques that may not be available or feasible in the kernel. A second consequence of our decision to not open the split mode driver's cross-domain interface to other users is that it cannot be reused, for instance by other drivers or user mode programs. This restriction is intended, as the generated interfaces are highly customized for a specific partitioning of a specific kernel component and any reuse beyond that use case bears a risk of misuse with fatal consequences.

7 Conclusion

Although microkernel OSs provide better isolation than monolithic OSs and modern implementations no longer suffer from the poor IPC performance of their ancestors, monolithic OSs still dominate the commodity desktop, mobile and server markets because of legacy code reuse and user familiarity. In order to benefit from both the existing code bases of monolithic systems and the design advantages of microkernel architectures, approaches to move kernel code portions of monolithic OSs into user mode have been proposed. While these approaches provide the mechanisms for split mode user/kernel operation of monolithic kernel code, they do not provide guidance on *what* code to execute in which mode. To this end, we propose a partitioning approach that combines static and dynamic analyses to assess the impact of kernel code partitioning decisions on both the degree of isolation and the expected performance overheads. Using collected data from profiling runs, we derive solutions that are *optimal* with respect to a user-defined isolation/performance prioritization.

We implement the approach for Microdrivers, an automated code partitioning framework for Linux kernel code, and demonstrate its utility in a case study of two widely used device drivers and a file system. Our results show that augmenting static analyses with data obtained from dynamic analyses allows to estimate the performance impact and, therefore, the feasibility of a whole spectrum of possible partitionings for production usage even before a split mode version is synthesized.

For future work, we plan to address the shortcomings of existing code partitioning tools encountered during the implementation of our profiling-based partitioning and add

support for atomic access operations and user mode interrupt handling. Furthermore, we plan to investigate more efficient alternatives for the complex function wrapping and parameter marshaling of Microdrivers to improve performance and the amount of movable functions.

ACKNOWLEDGMENTS

The authors would like to thank Matthew Renzelmann, Vinod Ganapathy, and Michael Swift for providing access to their Microdrivers framework.

REFERENCES

[1] J. Liedtke, "On μ-Kernel Construction," in *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proc. of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. ACM, 2009.

[3] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[4] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault isolation for device drivers," in *Proc. of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '09, June 2009.

[5] B. Döbel and H. Härtig, "Who Watches the Watchmen? - Protecting Operating System Reliability Mechanisms," in *Proc. of the Eighth USENIX Conference on Hot Topics in System Dependability*, ser. HotDep'12. USENIX Association, 2012.

[6] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum, "Keep net working - on a dependable and fast networking stack," in *Proc. of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '12, June 2012.

[7] J. Corbet and G. Kroah-Hartman, *Linux Kernel Development: How Fast It is Going, Who is Doing It, What They Are Doing and Who is Sponsoring the Work*, 25th ed. The Linux Foundation, August 2016.

[8] H. M. Walker, *The Tao of Computing*, 2nd ed. Chapman & Hall/CRC, 2012.

[9] J. M. Rushby, "Design and Verification of Secure Systems," in *Proc. of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. ACM, 1981.

[10] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors," in *Proc. of the 11th Workshop on ACM SIGOPS European Workshop*, ser. EW 11. ACM, 2004.

[11] M. Engel and B. Döbel, "The Reliable Computing Base: A Paradigm for Software-Based Reliability," in *Workshop on Software-Based Methods for Robust Embedded Systems*, 2012.

[12] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther, "The SawMill Multiserver Approach," in *Proc. of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, ser. EW 9. ACM, 2000.

[13] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The Design and Implementation of Microdrivers," in *Proc. of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII, 2008.

[14] Free Software Foundation, "GLPK (GNU Linear Programming Kit)." [Online]. Available: https://www.gnu.org/software/glpk/

[15] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," in *Proc. of the 12th USENIX Security Symposium*, ser. USENIX Security '03. USENIX Association, 2003.

[16] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proc. of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept 1975.

[17] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proc. of the 19th USENIX Security Symposium*, ser. USENIX Security '10. USENIX Association, 2010.

[18] D. Kilpatrick, "Privman: A Library for Partitioning Applications," in *Proc. of the FREENIX Track: 2003 USENIX Annual Technical Conference*, 2003.

[19] D. G. Murray, G. Milos, and S. Hand, "Improving Xen Security Through Disaggregation," in *Proc. of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. ACM, 2008.

[20] A. Mettler, D. Wagner, and T. Close, "Joe-E: A Security-Oriented Subset of Java," in *Proc. of 17th Annual Network and Distributed System Security Symposium*, ser. NDSS '10, 2010.

[21] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor," in *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011.

[22] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A Two-way Sandbox for x86 Native Code," in *Proc. of the 2014 USENIX Annual Technical Conference*, ser. USENIX ATC '14. USENIX Association, 2014.

[23] D. Brumley and D. Song, "Privtrans: Automatically Partitioning Programs for Privilege Separation," in *Proc. of the 13th USENIX Security Symposium*, ser. USENIX Security '04, 2004.

[24] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-privilege Compartments," in *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. USENIX Association, 2008.

[25] B. Jain, C.-C. Tsai, J. John, and D. E. Porter, "Practical Techniques to Obviate Setuid-to-root Binaries," in *Proc. of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. ACM, 2014.

[26] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. ACM, 2015.

[27] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[28] A. Cimitile and G. Visaggio, "Software Salvaging and the Call Dominance Tree," *Journal of Systems and Software*, vol. 28, no. 2, pp. 117–127, Feb. 1995.

[29] A. van Deursen and T. Kuipers, "Identifying Objects Using Cluster and Concept Analysis," in *Proc. of the 21st International Conference on Software Engineering*, ser. ICSE '99. ACM, 1999.

[30] P. Tonella, "Concept Analysis for Module Restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr 2001.

[31] S. Shaw, M. Goldstein, M. Munro, and E. Burd, "Moral Dominance Relations for Program Comprehension," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 851–863, Sept 2003.

[32] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st ed. Prentice-Hall, Inc., 1979.

[33] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Software Re-Modularization Based on Structural and Semantic Metrics," in *Proc. of the 17th Working Conference on Reverse Engineering*, ser. WCRE '10.

[34] T. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization," in *Proc. of the Fourth Working Conference on Reverse Engineering*, ser. WCRE '97, Oct 1997.

[35] T. C. Lethbridge and N. Anquetil, "Approaches to Clustering for Program Comprehension and Remodularization," in *Advances in Software Engineering*, H. Erdogmus and O. Tanir, Eds. Springer-Verlag New York, Inc., 2002, pp. 137–157.

[36] C.-H. Lung, M. Zaman, and A. Nandi, "Applications of clustering techniques to software partitioning, recovery and restructuring," *Journal of Systems and Software*, vol. 73, no. 2, pp. 227–244, 2004.

[37] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov 2007.

[38] M. Harman, R. M. Hierons, and M. Proctor, "A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization," in *Proc. of the Genetic and Evolutionary Computation Conference*, ser. GECCO '02. Morgan Kaufmann Publishers Inc., 2002.

[39] B. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, March 2006.

[40] K. Praditwong, M. Harman, and X. Yao, "Software Module Clustering as a Multi-Objective Search Problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, March 2011.

[41] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proc. of the Sixth Conference on Computer Systems*, ser. EuroSys '11. ACM, 2011.
[42] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, Apr 2013.
[43] A. P. Miettinen and J. K. Nurminen, "Energy Efficiency of Mobile Clients in Cloud Computing," in *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010.
[44] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proc. of the eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01, 2001.
[45] D. Simpson, "Windows XP Embedded with Service Pack 1 Reliability," January 2003. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx
[46] A. Ganapathi, "Why Does Windows Crash?" UC Berkeley, Tech. Rep. CSD-05-1393, May 2005.
[47] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP Kernel Crash Analysis," in *Proc. of the 20th Conference on Large Installation System Administration*, ser. LISA '06. USENIX Association, 2006.
[48] M. Mendonça and N. Neves, "Robustness Testing of the Windows DDK," in *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07, 2007.
[49] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten Years Later," in *Proc. of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. ACM, 2011.
[50] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," in *Proc. of the 6th Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. USENIX Association, 2004.
[51] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi, "Solving the Starting Problem: Device Drivers As Self-describing Artifacts," in *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. ACM, 2006.
[52] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USENIX Association, 2006.
[53] L. Tan, E. Chan, R. Farivar, N. Mallick, J. Carlyle, F. David, and R. Campbell, "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support," in *Proc. of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, ser. DASC '07, Sept 2007.
[54] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device Driver Safety Through a Reference Validation Mechanism," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. USENIX Association, 2008.
[55] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast Byte-granularity Software Fault Isolation," in *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. ACM, 2009.
[56] H. Jo, H. Kim, J.-W. Jang, J. Lee, and S. Maeng, "Transparent Fault Tolerance of Device Drivers for Virtual Machines," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1466–1479, Nov 2010.
[57] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software Fault Isolation with API Integrity and Multi-principal Modules," in *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011.
[58] R. Nikolaev and G. Back, "VirtuOS: An Operating System with Kernel Virtualization," in *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013.
[59] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," in *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, 2003.
[60] Y. Sun and T. c. Chiueh, "Side: Isolated and efficient execution of unmodified device drivers," in *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '13, June 2013.
[61] A. Kantee, "Rump file systems: Kernel code reborn," in *Proc. of the 2009 USENIX Annual Technical Conference*, ser. USENIX'09, 2009.
[62] G. Kerneis, "CIL (C Intermediate Language)." [Online]. Available: https://github.com/cil-project/cil
[63] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Conference on Compiler Construction*, 2002.
[64] M. J. Renzelmann and M. M. Swift, "Decaf: Moving Device Drivers to a Modern Language," in *Proc. of the 2009 USENIX Annual Technical Conference*, ser. USENIX '09. USENIX Association, 2009.
[65] S. Butt, V. Ganapathy, M. M. Swift, and C.-C. Chang, "Protecting Commodity Operating System Kernels from Vulnerable Device Drivers," in *Proc. of the Annual Computer Security Applications Conference*, ser. ACSAC '09, 2009.
[66] B. Ryder, "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, May 1979.
[67] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP'97 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer Berlin Heidelberg, 1997, vol. 1241, pp. 220–242.
[68] W. Gong and H.-A. Jacobsen, "ACC: The AspeCt-oriented C Compiler." [Online]. Available: https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc
[69] R. Love, "Sleeping in the interrupt handler." [Online]. Available: http://permalink.gmane.org/gmane.linux.kernel.kernelnewbies/1791
[70] "VFIO - "Virtual Function I/O"." [Online]. Available: https://www.kernel.org/doc/Documentation/vfio.txt
[71] "RTwiki." [Online]. Available: https://rt.wiki.kernel.org/index.php/Main_Page

Oliver Schwahn is a Ph.D. student at the Department of Computer Science at Technische Universität Darmstadt, Germany. His research interests include the design, analysis, and assessment of dependable software and systems.

Stefan Winter has obtained a doctoral degree in Computer Science from TU Darmstadt in 2015, where he is now working as a postdoctoral research fellow. His research interests span a variety of topics related to the design and analysis of dependable software systems from operating system design to system-level test efficiency.

Nicolas Coppik is a Ph.D. student at the Department of Computer Science at Technische Universität Darmstadt, Germany. His research interests include the design, analysis, and assessment of secure software and systems.

Neeraj Suri received his Ph.D. from the UMass-Amherst and is currently a Chair Professor at TU Darmstadt, Germany. His research addresses the design, analysis and assessment of trustworthy systems and software.