Practical Safe Linux Kernel Extensibility

Samantha Miller
University of Washington

Kaiyuan Zhang
University of Washington

Danyang Zhuo
University of Washington

Shibin Xu
University of Washington

Arvind Krishnamurthy
University of Washington

Thomas Anderson
University of Washington

ABSTRACT

The ability to extend kernel functionality safely has long been a design goal for operating systems. Modern operating systems, such as Linux, are structured for extensibility to enable sharing a single code base among many environments. Unfortunately, safety has lagged behind, and bugs in kernel extensions continue to cause problems. We study three recent kernel extensions critical to Docker containers (Overlay File System, Open vSwitch Datapath, and AppArmor) to guide further research in extension safety. We find that all the studied kernel extensions suffer from the same set of low-level memory, concurrency, and type errors. Though safe kernel extensibility is a well-studied area, existing solutions are heavyweight, requiring extensive changes to the kernel and/or expensive runtime checks. We then explore the feasibility of writing kernel extensions in a high-level, type safe language (i.e., Rust) while preserving compatibility with Linux and find this to be an appealing approach. We show that there are key challenges to implementing this approach and propose potential solutions.

CCS CONCEPTS

• **Computer systems organization → Reliability**; • **Software and its engineering → Language features**.

KEYWORDS

operating systems, extensibility, Rust

ACM Reference Format:
Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. 2019. Practical Safe Linux Kernel Extensibility. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321429

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321429

1 INTRODUCTION

Extensibility has long been a desired property of operating systems as it allows users with diverse needs to share the same core operating system codebase. This concept has been widely incorporated, and Linux interfaces are designed to be easily extensible. Today, kernel extensions provide critical functionality to ensure security and efficiency in the cloud. For example, Docker containers [8] rely on Open vSwitch Datapath [22], AppArmor [2], and OverlayFS [21]. Bugs in these extensions can affect the containers using them, the user's containerized applications, and potentially the rest of the system. Ensuring the correctness of extensions is therefore necessary for the stability of modern cloud solutions.

Despite significant work in the area of safe extensibility, such as VINO [24], SFI [27], and SPIN [3], existing solutions are too heavyweight for modern operating systems, requiring either expensive changes to the kernel and/or expensive runtime checks. More lightweight solutions, such as languages like the Berkeley Packet Filter [18], do not support the full complexity of kernel extensions. In Linux, the main operating system used in the cloud, many kernel extensions do not incorporate any of these existing solutions and remain unprotected. Bugs in these unprotected extensions therefore have the potential to affect the integrity of the underlying system. Therefore, low-cost kernel extension safety is needed to increase the stability of cloud systems.

We conduct a bug analysis of three kernel extensions (Open vSwitch Datapath, AppArmor, and OverlayFS) in order to understand the properties of errors in Linux extensions. We focus on bugs that can be caught without knowledge of the specific extension's semantics and find that many of these bugs relate to memory, concurrency, or typing. Linux does little to isolate errors caused by kernel extensions, and these bugs can cause undesirable behavior in the kernel (e.g., panics, memory leaks, kernel thread crashes, deadlocks). Moreover, 35% of the analyzed bugs involve multiple kernel-to-extension transitions, and reasoning about this set of bugs requires understanding the workflow between the kernel and the extensions.

We explore the feasibility of writing kernel extensions in a type-safe, memory-safe, race-free language (i.e., Rust) while maintaining compatibility with Linux. While using a high-level, type-safe language for extensibility isn't new [3, 7, 13, 15], previous works rewrite the whole operating system. That approach doesn't provide the interoperability and ease of deployment that incremental extensibility requires. We propose writing only extensions in the type-safe language. This raises several questions: Can this approach provide reasonable guarantees, and if so, what must be done to achieve these guarantees? We find that Rust can provide the desired properties, preventing 93% of analyzed bugs. We also explore rewriting a simple file system—RomFS [23]—into Safe Rust while maintaining compatibility with unmodified Linux.

While porting RomFS, we discovered challenges to supporting this approach. First, Linux kernel interfaces are not designed for compile-time safety checks. Linux interfaces rely heavily on pointers that obscure type information. Since Rust relies on its type system for safety, a naive implementation would not provide the full safety benefits. Second, the hybrid code flow, where the code path repeatedly switches between kernel and kernel extensions, causes challenges, particularly for Rust's resource management. Linux kernel extensions commonly employ design patterns that involve passing resources across the OS/extension boundary, but this pattern complicates Rust's automatic memory management. Finally, translating C to Rust has general difficulties. Kernel extensions rely heavily on pointer arithmetic and type casting, which are restricted in Rust for safety. In this paper, we explore these issues further and propose possible solutions. We believe that this approach has the potential to provide incremental extensibility for Linux.

2 EXISTING ERRORS

To understand how to increase the safety of kernel extensions, we analyzed the causes of errors found in several existing Linux kernel extensions. We split bugs into two categories: high-level and low-level. High-level bugs are errors in extension semantics and require extension-specific information to detect and debug. In contrast, low-level bugs do not require specific knowledge and are always bugs, such as a `NULL` pointer dereference or an allocate without a free. High-level bugs make up 50% of the analyzed bugs, but are out of scope for this project and are left for future work.

Since our primary focus is on cloud systems, we analyzed kernel extensions used to support Docker containers. For diversity of behavior, we chose extensions related to security, networking, and file systems. Respectively, the relevant kernel extensions are AppArmor, Open vSwitch Datapath, and OverlayFS. We analyzed the commit history of each extension from 2014 to 2018. For each bug fix commit, we determined the cause and effect of the bug.

In this analysis, we seek to provide information about the general properties of bugs in kernel extensions. Our analysis is limited by the small number of extensions we studied and our method for discovering bugs. Since we rely on the commit history, we only count bugs that have been found and fixed. While our analysis is not a comprehensive bug study, we identify broad categories of bugs that currently exist in the selected extensions. We believe this information can be used to guide future research in Linux extensibility.

We first describe how Linux handles errors. We then quantify the distribution and effects of the analyzed bugs.

2.1 Linux Error Handling

Linux does relatively little to isolate the kernel from extensions. Once an extension is loaded, it is treated as part of the kernel and is given the same level of trust as the rest of the kernel. Because of this, an extension can arbitrarily modify any accessible kernel state and can be responsible for managing memory of kernel data structures. Errors caused by the kernel extension are handled in the same way as errors from any other part of the kernel.

The Linux kernel as a whole tries to limit the effects of errors on the system. Panics are relatively rare, and their usage is generally discouraged. More commonly error paths are `oops` and `BUG`. A kernel `oops` is a common error path for issues in the Linux kernel. The kernel prints a stack trace and tries to handle the issue if possible, `panic`ing if the issue is catastrophic. `Oops`es are caused by several types of low-level bugs, such as `NULL` pointer dereferences and bad page faults. An `oops` normally kills the offending kernel process, but can panic when inside an interrupt (or, unlikely, if the kernel is configured to panic on an `oops`). The offending kernel thread is not given the chance to clean up, and the kernel can be left in an inconsistent state. The `BUG` and `BUG_ON` functions act as asserts in the kernel. When one of these is hit, the kernel dumps register and stack information and kills the kernel process. An extension usually causes a `BUG` by freeing a junk value or potentially by executing a double free. Like a kernel `oops`, a `BUG` can leave the kernel in an inconsistent state.

2.2 Types of Errors

We defined three broad categories of low-level bugs: memory, concurrency, and type. Memory bugs relate to incorrect memory usage, such as `NULL` pointer dereferences, use-after-free, and memory leaks. Concurrency bugs involve locks or data races. Type bugs result from a mismatch between the variable's type and how it is used. Often, this occurs because

Bug	Number	Effect on Kernel
Use Before Allocate	6	Likely `oops`
Double Free	4	Undefined
`NULL` Dereference	5	`oops`
Use After Free	3	Likely `oops`
Over Allocation	1	Overutilization
Out of Bounds	4	Likely `oops`
Dangling Pointer	1	Likely `oops`
Missing Free	18	Memory Leak
Reference Count Leak	7	Memory Leak
Other Memory	1	Variable
Deadlock	5	Deadlock
Race Condition	5	Variable
Other Concurrency	1	Variable
Unchecked Error Value	5	Variable
Other Type Error	8	Variable

Table 1: Count of analyzed bugs with effects of each bug, categorized as memory, concurrency, or type. Effects are described in subsection 2.1.

error values were treated as valid data. All of the analyzed bugs fit into one of these three categories.

Table 1 shows the prevalence and effects of these bugs in the kernel extensions we studied. Some bugs have "variable" effects because the specific bug doesn't always lead to the same consequence. For example, a race condition on a reference count can cause either a memory leak or a use-after-free, depending on the exact interleaving of operations. Most (68%) of the bugs we saw were memory bugs, and many of these memory bugs (50%) were a form of memory leak. An `oops` is the likely outcome of 26% of the bugs, while other bugs have the potential to cause an `oops`; the "variable" effect bugs can cause an `oops`, often because unknown pointers are dereferenced. While memory leaks do not affect the immediate functioning of the kernel, that memory is lost from the system and can affect kernel operation unrelated to the extension. Like the inconsistent state caused by an `oops`, a memory leak cannot be fixed without restarting the machine.

We noticed that many of the bugs appear in error handling pathways or occur because of incorrect handling of error values. We posit that it is difficult to test for bugs in error handling code, so these errors are harder to catch by standard testing infrastructure.

We found it helpful to also consider the resource lifetime of some of the bugs. Both memory allocation and locking involve a time frame when the relevant resource is needed. Acquiring or releasing the resource at the wrong time often results in bugs, such as deadlocks, memory leaks, use-after-free bugs, or double frees. We classify these bugs by whether the resource should only be held within one kernel call into the extension or across multiple kernel calls. We found that 35% of relevant bugs involved behavior that spanned multiple kernel calls while the rest involved resources held within one kernel call.

3 CURRENT APPROACHES

There has been significant work in the space of safe extensibility. We classify previous projects into five categories.

High-level language. Operating systems such as Spin [3], Singularity [13], Biscuit [7], and TockOS [15] are written in high-level, type-safe languages. Spin, Singularity, and Biscuit use garbage collected languages, while TockOS uses Rust to provide safety without the added complexity of kernel garbage collection. These solutions increase the safety of kernel code using the memory and type safety properties of the languages they employ. Since these guarantees depend on memory and type safety, many of the bugs analyzed above would be impossible. These solutions require writing new operating systems, and therefore do not provide incremental extensibility to Linux. We can still use the knowledge from these projects to influence our design. For example, TockOS also faces challenges writing kernel data structures in Safe Rust and integrating C and Rust.

Interpreted/JIT language. Languages such as the Berkeley Packet Filter (BPF) [18], or more recently the extended Berkeley Packet Filter (eBPF), provide safety guarantees by limiting the potential behavior of the extension. This approach allows user programs to run in an in-kernel virtual machine, verifying safety properties and restricting permissions when the code is loaded. eBPF does provide efficient incremental extensibility, but imposes significant restrictions on extensions. For example, eBPF disallows dynamic loops and limits the maximum number of instructions. In addition, eBPF programs must fit within one of the provided program types which define what permissions the program has. eBPF primarily supports networking functionality, and there has been work implementing Open vSwitch Datapath using eBPF [26], but given the restrictions, it is difficult to imagine this approach being feasible for general extensions.

Software fault isolation. Software fault isolation mechanisms [4, 11, 17, 24, 27] apply runtime checks to ensure that faults in extensions cannot affect the rest of the system. The runtime checks needed to guarantee safety are extensive, and even the relatively small performance overhead is more than modern systems are willing to accept. In addition, SFI does not solve the same problem that we would like to address. SFI assumes that kernel extensions can be malicious, so it focuses on restricting execution of extensions. We do not address malicious code and instead focus on code quality, helping well-meaning developers write correct code.

Userspace. Another approach to extension safety is to move extensions to userspace to limit their permissions. Systems with minimal kernels, such as the Exokernel [10] and microkernel-based systems [1, 16], require this for all kernel extensions. Some Linux kernel extensions can be structured this way, such as implementing a file system using FUSE [12] or implementing drivers in userspace [9]. However, this approach complicates development and can add performance overhead. Since the in-kernel and system call interfaces are so different, it is not obvious that all extensions, especially security extensions, could be implemented in userspace.

Verification. One method of ensuring code safety is verification. There has been recent work [14, 20, 28] verifying correctness of operating systems and file systems [5, 6, 25]. In one work focused on extensibility [19], kernel extensions are verified, and the kernel requires the appropriate verification when the extension is loaded. This approach can provide strong safety guarantees, theoretically preventing all bugs (including high-level bugs not addressed in this work), but requires high development effort. There has been work on reducing the burden of verifying operating systems [20, 28], but we do not believe that verification is mature enough yet to be widely accessible. Currently, employing verification requires a significant time investment and expert knowledge beyond what most developers have.

4 INCREMENTAL EXTENSIBILITY WITH RUST

We explore a design that adapts the high-level language approach to provide incremental extensibility. In our proposed approach, developers can write kernel extensions in Rust without modifying the rest of the Linux kernel. We believe that this approach has the potential to provide a practical solution to safe incremental extensibility.

In our design, kernel extensions can be written purely in Safe Rust, the subset of the language that does not allow pointer dereferences. We chose Rust because it provides zero cost abstractions and interoperates well with C. Like Modula-3 used in Spin [3], Safe Rust provides strong safety guarantees using a strict type system, preventing 93% of the bugs we analyzed (only leaving deadlocks). Unlike Modula-3, Rust does not have a garbage collector and has little performance overhead compared to C. Rust has built in functionality to interoperate with C, so C code can remain untouched.

We propose a lightweight framework for implementing Linux kernel extensions in Safe Rust without modifying the existing Linux kernel. Our system is impremented as a shim layer between the kernel and the extension. This framework implements safe wrappers around unsafe kernel functions and enforces safe typing for the extension interface. The shim layer and the kernel are trusted and can therefore contain unsafe code.

```
static int romfs_fill_super(
    struct super_block *sb,
    void               *data,
    int                 silent
)
```

Figure 1: Function signature from RomFS showing common pointer use.

As a proof of concept, we have applied this approach to RomFS, a read-only, in-memory file system. RomFS is a simple extension, only containing ~700 lines of code. This endeavor brought our attention to some challenges with implementing Linux extensions in Rust. We describe these challenges and present possible solutions.

4.1 Challenges

While most code translation from C to Safe Rust has been straightforward, some kernel design patterns require new solutions. We identify two main sources of this difficulty: kernel interfaces and the hybrid code flow involving code in multiple languages.

Interfaces. Kernel interfaces, while designed for extensibility, are not designed for type safety. Rust relies on its strong type system to provide safety, however kernel interfaces use techniques that obscure type information. All data structures and unstructured memory regions are given to the extension as raw pointers. Pointers contain no built-in guarantees about the size or validity of the region they point to, so cannot be dereferenced in Safe Rust. This is a standard mechanism of passing information in C, but isn't compatible with Rust's safety checks. Without changes to the OS/extension interface, the benefit of using Rust will be severely limited.

For example, file systems in the Linux kernel implement a function that fills the kernel's `super_block` data structure. The RomFS function signature, using the kernel's defined interface, is provided in Figure 1. The kernel passes in two pointers, one to the `super_block` data structure and the other to a blob of file-system specific `data`. In this function, the file system is expected to fill in the `super_block` data structure, potentially using the extra `data`. Both of these pointers are challenging to incorporate in Safe Rust. Writing to the super block would require dereferencing a raw pointer. The `data` variable is even more difficult. While the `super_block` has type information provided by the kernel, `data` is completely opaque. In practice, this points to a string containing options for the file system, but this type information is obscured.

```
struct romfs_inode_info {
    struct inode        vfs_inode;
    struct              metadata;
};
```

Figure 2: Internal RomFS `inode` data structure.

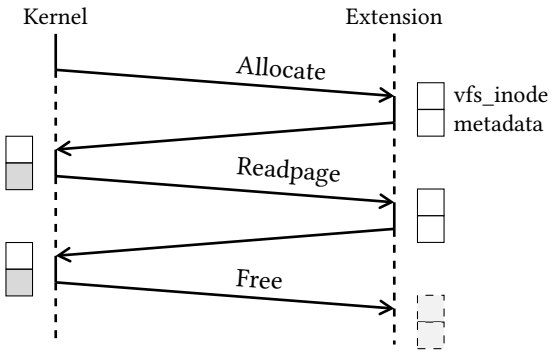

Figure 3: `romfs_inode_indo` control flow across the kernel-to-extension boundary.

Hybrid Code Flow. Another fundamental challenge stems from hybrid code flow—one code path involving multiple languages. As described in subsection 2.2, extensions currently contain a significant number of bugs across the OS/extension boundary, and any solution must address this problem. So far, we have primarily analyzed resource management, specifically memory management. Rust's memory model depends on ownership and lifetimes. A resource owns the memory region of its data. Memory is allocated when the resource is created, and when the resources goes out of scope, the resource's lifetime ends and the memory is freed. This model cleanly manages memory for resources that live purely in Rust. However, some resources in our system must live across the language boundary.

For example, RomFS allocates memory for an internal structure called the `romfs_inode_info`, shown in Figure 2, which contains the kernel `inode` as the `vfs_inode` field. The general control flow for this data structure is shown in Figure 3. When an `inode` is needed, the kernel requests one from the extension. RomFS allocates a `romfs_inode_info` and returns the pointer, as an `inode` pointer, to the kernel. The kernel later passes the `inode` pointer back to the extension, and the extension assumes that this inode is part of a `romfs_inode_info` and uses pointer arithmetic to access the metada. When the `inode` is no longer needed, the kernel instructs RomFS to free the `inode`, and the extension frees the `romfs_inode_info`. This API is defined by the Virtual Filesystem Switch (VFS) interface and is commonly used in file system extensions. This design pattern allows the extension to extend a kernel data structure, and this model appears in every extension we have analyzed.

Even ignoring the unsafe pointer arithmetic, this model is difficult to support in Rust. Since the kernel is written in C, the Rust compiler cannot track the ownership of the `inode` and cannot determine when its lifetime ends. Even if the Rust compiler could be instructed to track the `inode` ownership across the OS/extension boundary, the problem still remains for the `romfs_inode_info` data structure. Since this data structure is only accessed using pointer arithmetic, the Rust compiler is given no indication that the data is still accessible and should not be freed.

4.2 Proposed Solutions

We focus on solutions that require no changes to the Linux kernel and few changes to the structure of the extension. Our proposals below could be implemented as Rust code around the extension or between the kernel and the extension. An extension developer would use the provided code when applicable but would otherwise see few changes.

Interfaces. Our proposed solution to interface safety is to add type information where possible (such as for `data` above, and translate pointers provided by the kernel into capabilities to access the relevant kernel resources. We trust that the kernel is passing correct pointers to the extension so the generated capabilities are guaranteed to be valid. Unlike pointers, the extension cannot create capabilities or modify their locations using pointer arithmetic. For example, the `struct super_block` pointer in Figure 1 would be converted into a Rust `SuperBlock` type, and the extension would use provided getters and setters to interact with it. The getter and setter functions are implemented within the framework's shim layer and are most often automatically generated using provided macros. Non-trivial conversions are implemented using manually written functions, but further automatic generation is preferred for future work.

Hybrid Code Flow. Our work on addressing the hybrid code flow has so far focused on memory management. We suggest three memory primitives for the extension: global variables, kernel-call local variables, and kernel-object bound variables. Global variables are supported by default in Rust, and no extra support is needed. Kernel-call local variables are standard Rust data structures and are appropriate for any resource that exists within one call from the kernel into the extension. The lifetime can be tracked by the Rust compiler, and the memory will be automatically freed when the variable goes out of scope. We introduce a `MemContainer` type to represent unstructured regions of memory, such as buffers.

The `MemContainer` can be created from kernel heap memory, using `kmalloc` when created and freeing the allocated memory when the `MemContainer` goes out of scope.

In our experience, extension resources held across kernel calls, such as RomFS's `romfs_inode_info`, are always associated with a kernel data structure, in this case the kernel `inode`. As shown in Figure 3, the memory for this data structure is allocated and deallocated by the extension, but managed by the kernel through explicit calls to the extension. We propose modeling this in Rust by binding the lifetimes of extension data structures and the kernel data structures. This means that when a kernel data structure is freed, any associated extension data structures will be automatically freed. We trust that the kernel correctly indicates when to free these data structures, so memory leaks across multiple kernel calls should be impossible. In RomFS, this involves tracking the ownership of `inode`s through the hybrid code flow shown in Figure 3. For example, after the `romfs_inode_info` is allocated, ownership of the `inode` is passed to the kernel. For `Readpage`, the kernel passes an `inode` reference to the extension. When `Free` is called, ownership of the data structure is passed to the extension, and the `romfs_inode_info` is freed.

5 DISCUSSION

We have so far only explored the feasibility of our approach and have not developed a full solution. Synchronization techniques used within the Linux kernel are critical areas for further research. Kernel extensions can use kernel locks and can therefore have resource management bugs associated with those locks. While Rust does not guarantee deadlock freedom, deadlocks are serious issues.

Converting C code into Safe Rust requires effort regardless of the use case. Some kernel design patterns make wide use of pointer arithmetic, and these patterns must be redesigned for our approach (see Section 3.1). Using mutable global variables is also difficult in Rust because they require synchronization for safety. These issues are not particular to kernel extensibility, but are general drawbacks of programming in Rust. Other projects that propose writing kernels in Rust [15] have encountered and addressed these issues. We intend to this knowledge to guide our solution.

We believe incremental extensibility has the potential to change the nature of kernel extensions. By increasing the safety of extension code, this approach can allow developers to write kernel extensions that would previously be userspace programs. Applications could then take advantage of the potential performance gains and flexibility of being kernel extensions. Containers are popular, and adding container features typically requires kernel modification. Our approach could be an attractive option for implementing advanced container features in Linux.

6 CONCLUSION

Safe kernel extensibility has been a desired property of operating systems, and has been a target of significant research. Extensibility is widely used, but extension safety has lagged behind. Extension safety is especially important today due to the extensive use of containers in the cloud. Existing solutions are heavyweight and add significant development cost or runtime overhead.

We analyze the current state of Linux extension errors and explore the feasibility of using a high-level, type-safe language to allow developers to write Linux kernel extensions in Rust. We identify key challenges to this approach and propose possible solutions. Based on our experience, this approach has the potential to provide attractive safety and performance benefits to Linux extensions.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation For UNIX Development. In *Summer USENIX*.

[2] AppArmor 2018. AppArmor. https://www.kernel.org/doc/Documentation/security/apparmor.txt. (2018).

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*.

[4] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-granularity Software Fault Isolation. In *SOSP*.

[5] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *SOSP*.

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *SOSP*.

[7] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In *OSDI*.

[8] Docker 2018. Docker. https://www.docker.com/. (2018).

[9] DPDK 2017. Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide. https://doc.dpdk.org/guides/prog_guide/. (2017).

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*.

[11] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *OSDI*.

[12] Fuse 2018. Filesystem in Userspace. https://github.com/libfuse/libfuse. (2018).

[13] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS OSR* (2007).

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*.

[15] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *SOSP*.
[16] Jochen Liedtke. 1995. On Microkernel Construction. In *SOSP*.
[17] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Software Fault Isolation with API Integrity and Multi-principal Modules. In *SOSP*.
[18] Steven McCanne and Jacobson Van. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX*.
[19] George C. Necula and Peter Lee. 1996. Safe Kernel Extensions Without Run-time Checking. In *OSDI*.
[20] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP*.
[21] Overlayfs 2018. Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. (2018).
[22] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open vSwitch *(NSDI)*.
[23] romfs 2019. ROMFS - ROM FILE SYSTEM. https://www.kernel.org/doc/Documentation/filesystems/romfs.txt. (2019).
[24] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*.
[25] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button Verification of File Systems via Crash Refinement. In *OSDI*.
[26] William Tu, Joe Stringer, Yifeng Sun, and Wei Yi-Hung. 2018. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumbers Conference*.
[27] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *SOSP*.
[28] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *PLDI*.