An Incremental Path Towards a Safer OS Kernel

Jialin Li
University of Washington

Samantha Miller
University of Washington

Danyang Zhuo
Duke University

Ang Chen
Rice University

Jon Howell
VMware Research

Thomas Anderson
University of Washington

Abstract

Linux has become the de-facto operating system of our age, but its vulnerabilities are a constant threat to service availability, user privacy, and data integrity. While one might scrap Linux and start over, the cost of that would be prohibitive due to Linux's ubiquitous deployment. In this paper, we propose an alternative, incremental route to a safer Linux through proper modularization and gradual replacement module by module. We lay out the research challenges and potential solutions for this route, and discuss the open questions ahead.

CCS Concepts

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

Keywords

kernel safety, verified systems, reliable systems

ACM Reference Format:
Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. 2021. An Incremental Path Towards a Safer OS Kernel. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465277

1 Introduction

This is a call to arms to evolve a widely used operating system into one that is also safer and functionally correct.

Linux, through a commitment to open source and a single code base, has become the de-facto standard operating system of our age: the foundation of everything from smart devices to smart phones to routers to servers.

To accommodate this increasing scope, Linux developers have been adding over 1.5M lines of new code per year, to a codebase that already stretches to several tens of millions of lines. The result is not that surprising: hundreds of new security vulnerabilities are reported each year, and lifetime analysis suggests that the new code added this year has introduced tens of thousands of more bugs.

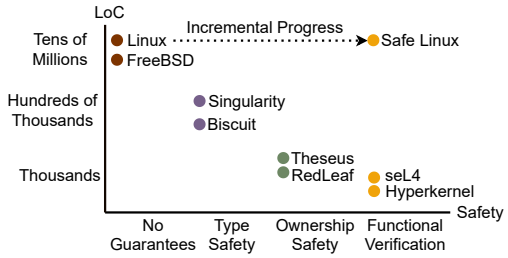


Figure 1: Our vision and the current state of systems.

While operating systems are far from the only source of security vulnerabilities, it is hard to envision building trustworthy computer systems without addressing operating system correctness.

One attractive option is to scrap Linux and start over. Many past works focus on building more secure and correct operating systems from the ground up: ones based on strong typing [22, 32, 48, 50], ones based on even stronger models such as linear types [13, 44], and ones that formally prove correctness for all or parts of a kernel [19, 35, 46, 51]. These OS kernels have significantly fewer features than Linux (Figure 1), impeding adoption.

There is another path, enabled by the development of better tools, better languages, and better verification systems. Instead of scrapping Linux, can we improve it incrementally?

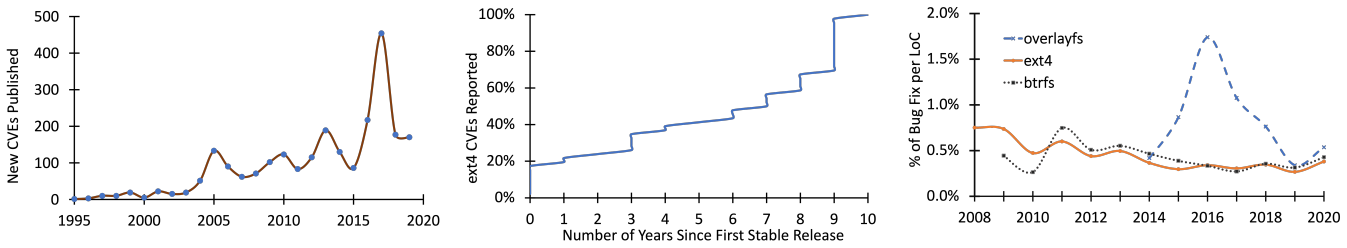
The common design patterns used in Linux development do not make incrementalism easy. Typically, Linux kernel modules interact through shared data structures with poorly specified locking constraints. The boundary and functionality of modules also lack clear separation. These patterns make integrating safe components difficult. In the following sections, we first examine the current state of bugs in Linux, and then propose a roadmap to incrementally make Linux safer. We then list research challenges implied by these design patterns and our suggested approaches. Lastly, we discuss the state of the art in systems verification and other related work.

2 Motivation

As with other commercial systems, Linux tries to strike a balance between growth and reliability. A typical release cycle consists of a two-week merge window and eight or more

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465277



(a) New Linux CVEs **(b) Reported ext4 CVEs** **(c) Number of bugs per LoC in file systems**

Figure 2: Bug analysis for Linux and its file systems. (a) The number of new CVEs reported each year. (b) The CDF of when CVEs in ext4 were reported after its initial release. (c) The number of bugs per line of code per year in various Linux file systems.

weeks of bug fixes and stabilization. During testing periods, developers are encouraged to test changes through static analysis tools for C [2, 9], automated testing frameworks [1, 4, 6], and continuous integration tests [5, 7]. However, even after years of rigorous testing and development, hundreds of common vulnerabilities and exposures (CVEs) are still found in Linux each year (Figure 2a). One may hope that vulnerabilities come from newer components that become reliable as they stabilize. However, when we look at ext4, a Linux file system in wide use for 12 years, 50% of CVEs in ext4 were found after 7 years or more of use (Figure 2b). Other Linux file systems share a similar trend. We show the number of new bug patches in overlayfs, ext4, and btrfs in Figure 2c since their initial releases. Even after 10 years, there are still new bugs (0.5% bugs per line of code each year) in all three file systems.

Despite efforts in static analysis tools for C and automated testing frameworks, Linux still struggles with numerous bugs. Properties like data race freedom and semantic correctness are difficult to exhaustively test but can be achieved through safe languages and formal verification. Over the past decade, the overhead and limitations of using safe languages and formal verification have been significantly reduced due to the development of new programming language designs [26, 49], efficient constraint solvers [56], and increasingly mature verification toolchains [21, 36, 53].

We believe that static properties, including type safety, ownership safety, and functional correctness, can make Linux safer and enable more effective testing. Type safety prevents unsafe type conversion, such as void pointer casting commonly found in Linux. Ownership safety is type safety augmented with an ownership model that guarantees memory safety and thread safety. Functional correctness guarantees that a system behaves according to its specification. For example, a specification for a file system can describe user observable effects on underlying files for each operation.

Adopting these practices would make a substantial dent in the prevalence of bugs in released versions of Linux. We analyzed all Linux CVEs from 2010, categorizing their Common Weakness Enumeration IDs by which errors can or cannot be prevented by various techniques. Among the 1475 total CVEs we examined, roughly 42% CVEs could be prevented with compile-time type and ownership safety, and an additional 35% with functional correctness verification. The remaining 23% have a variety of causes: improper security designs such as weak access restriction or overexposing kernel information, numeric errors like integer overflow and underflow, and various other causes. Some of these bugs could be prevented with programming language techniques such as mandatory overflow checks, automatic iterators for arrays, and misuse of uninitialized variables; others could benefit from better techniques to verify security properties, but which are beyond the scope of this paper.

Prior work has shown that it is already possible to build clean-slate operating systems using these techniques to achieve various degrees of safety [35, 37, 44, 46]. Unfortunately, the cost of switching from Linux to these clean-slate designs is prohibitive due to the established Linux and Android ecosystems. This raises an important question: *can we use modern safe languages and formal verification techniques to improve OS kernel safety without resorting to a clean-slate OS design?*

3 Our Roadmap

Our high-level approach is to enable *incremental safety with incrementally safer interfaces*. Concretely, we propose adding incremental safety to Linux while retaining compatibility, along two axes: components can be replaced one at a time, and each component can be replaced with an incrementally-safer implementation. This idea has two costs: First, incrementally replacing modules requires modular interfaces, which can result in performance cost. Second, incremental changes require careful movements to be compatible with the existing code, increasing development effort. The benefit, however, is that each change adds immediate benefits to the kernel: that component now has a more robust implementation and can better support growth by resisting regressions.

Step 1: Modularity. As a first step, we propose introducing modular interfaces around existing Linux components. Specifically, callers of any module must only reference the modular interface and cannot directly depend on any specific

implementation. Modular interfaces allow each module to be improved incrementally without affecting others, and provide isolation and encapsulation, and enable easier reasoning for verification.

Step 2: Type safety. The second step is to introduce type safety to prevent type errors. A module is rewritten without the use of `void` pointers or casting values to incompatible types, such as casting error values to pointers. This can be accomplished by writing modules in C++ or another language with stronger-than-C typing.

Step 3: Ownership safety. Next, type safe modules are enhanced with ownership safety. Ownership safety is a multi-threaded version of memory safety. Where memory safety requires that there are no accesses to invalid memory, such as NULL pointer dereferences or use-after-free bugs, ownership safety adds that concurrent access to memory is also protected. Type safety plus ownership safety allows the module to know both that it is interpreting memory correctly and that it has the rights to access, mutate, or free that memory. Modules written in a type- and ownership-safe language such as Safe Rust are immune to entire classes of bugs, from NULL pointer dereferences to buffer overruns to memory leaks to data races, and are still able to perform complicated work with competitive performance [44].

Step 4: Functional correctness. Finally, a module is enhanced with partial or complete functional correctness verification. This requires developing a specification for some or all of the modules and ensuring that a module matches its specification. Functional correctness depends on ownership safety since module semantics is undefined under undefined memory access behavior. Functional correctness checks are able to prevent wide varieties of bugs, limited only by the depth of the specification.

Summary. Each step imposes greater restrictions on a module, and thus imposes different requirements on the interfaces that module must implement. Modular components need interfaces that abstract component behavior and isolate all functionality to the module. Type safety cannot be provided if `void` pointers are passed to the module. Memory safety requires that the caller grants memory rights to the callee and that the interface defines those rights. Functional correctness checks explicitly require specifications of expected interface behavior and can become intractable if interfaces are too complicated [31]. Each requirement strengthens the previous, so the interface for one step informs the interface for the next.

4 Research Challenges

The Linux kernel prioritizes flexibility and performance in its interface design. Static safety checks require human-writable and machine-understandable contracts at the interfaces. We analyze how this leads to four challenges for our roadmap and present proposed blueprints for interface designs.

4.1 Modularity

Challenge: Monolithic Structure. Because of its focus on performance, Linux often lacks strict module boundaries between components. Some Linux interfaces do enforce modularity; for example, VFS provides an abstract file system interface; other components like the network stack aren't cleanly separated. Extracting modular interfaces and fixing call sites is a necessary precondition for integrating statically checked modules.

Approach: Modular Interfaces. A modular interface should provide an abstract representation of module behavior but isolate its internals from other parts of the kernel. The kernel must explicitly call the module through the interface rather than arbitrarily. It would support various underlying implementations as long as they have the same interface. New implementations can be dropped in without changing other parts of the kernel. Such refactoring is not only beneficial for integrating safer modules, but also for allowing modules that provide other interesting properties, such as specialized performance goals or use of specialized hardware features. As an example, VFS didn't just happen; VFS was a response to the need to support new functionality–network file systems–alongside the existing native file system.

We recognize that modular interfaces can potentially prevent performance optimizations that rely on cross-module co-operation and can be difficult to introduce for subsystems that are not designed with modularity in mind. For example, while Linux sockets support multiple protocol families and multiple protocols within those families, references to TCP state can be found throughout generic socket code and data structures. Adding support for modular interfaces to subsystems like this will pose a challenge but could benefit both this project and others that propose alternative implementations for modules such as TCP stacks [34, 38, 42], to meet emerging demands and make efficient use of recent hardware advances.

This raises some interesting research questions: how do we retrofit modularity into subsystems that aren't designed for it? How do we introduce modularity while maintaining efficient and flexible interfaces?

4.2 Type Safety

Challenge: Type Confusion. Linux is written in C with no check to disallow arbitrary pointer casting. Casting pointers to incompatible types is common, with developers relying on assumptions or manual runtime checks to determine the correct types. For example, VFS allows a file system to pass custom data between `write_begin` and `write_end` by passing void pointers to the two functions. In `write_end` the file system assumes that the pointer was from its `write_begin` function and casts the pointer to the relevant type. Many functions, such as VFS `lookup`, return a pointer on success or an

error value on failure. To achieve this in C, the error value is cast to a pointer, and the caller must manually check that the pointer is valid before dereferencing it. Similar type confusion errors can be seen throughout the network stack: custom data gets wrongly casted and leads to denial of service [3]. Interfaces for type safe modules must be written so they don't require these types of unsafe casting.

Approach: Eliminate Unsafe Interfaces. The `void` pointers used to pass custom data structures can be replaced with pointers to a generic type using language-level techniques such as C++ templates or Rust generics. To eliminate the need for casting error values to pointers, type safe interfaces either pass a pointer argument for the error value to be written to or require functions to return a union type that can hold either valid data or an error, such as Rust's `Error`. Practical type confusion detection is another open research question. While type detection frameworks exist for languages like C++ [28], there is nothing robust and low overhead for Linux yet.

4.3 Ownership Safety

Challenge: Complex Ownership Sharing. The Linux kernel passes information between components by passing shared-memory data structures across the interface boundaries. These data structures are accessed concurrently by different sections of the kernel, often with complicated specifications on which fields can be accessed when, by which functions, and when which locks need to be held. These data structures are often passed as non-`const` pointers, so the only thing preventing incorrect access is vigilant code review.

For example, the kernel's generic inode data structure is passed from the VFS layer to the file system on most file system calls. Many of the inode's fields aren't associated with any inode-level synchronization mechanism since they're only modified on specific, known code paths protected by other synchronization mechanisms. Three fields are explicitly protected by the `i_lock` field, but one of those three, the `i_size` field, is only *maybe* protected, according to the relevant comment. File systems are responsible for updating `i_size`, so they must be able to determine the correct synchronization behavior. Additionally, only some code paths in the VFS layer will lock `i_lock` before calling into the file system, so the synchronization requirements are different depending on the function in the file system.

Since the kernel's interfaces don't include any information about the ownership rights, they can't be directly used to write modules with static ownership safety checks. Additionally, the complexity make it difficult to know what the correct ownership contract is for a shared data structure, and field-by-field ownership definitions that can change by call are difficult to encode in a framework for statically checked ownership safety.

Approach: Restricted and Explicit Ownership Sharing. We identify two key ideas for ownership safe module interfaces: restricted and explicit ownership sharing.

Restricted Ownership Sharing. Ownership safe modules should *restrict* the ways that memory is shared across module boundaries to enforce simple contracts that are easy to encode. We are inspired by message passing interfaces used for strict memory separation, such as in microkernels and across the kernel/userspace boundary. However, message passing interfaces are stronger than necessary and can impose performance overhead caused by memory copies. We propose interfaces that are semantically equivalent to message passing interfaces but share memory for performance reasons. Three models are:

(1) Memory ownership is passed. The caller can no longer access the memory. The callee must free the memory.
(2) Exclusive rights to the whole memory region are passed. The caller cannot access the memory until the call returns. The callee can mutate the memory but not free it and cannot access the memory after the call returns.
(3) Non-exclusive rights to the whole memory region are passed. The caller, callee, and others can read the memory, but none can mutate the memory until the call returns. Again, the callee cannot free the memory and cannot access the memory after the call returns.

In prior work, Bento allows Linux kernel file systems to be written in safe Rust by leveraging the FUSE low-level API with limited ownership sharing [43]. The FUSE API is designed to function across the kernel/userspace boundary, so Bento's interface is sufficient to satisfy the proposed interface model but is somewhat stronger than necessary.

Explicit Ownership Sharing. Ownership contracts to prevent unsafe access and resource allocation contracts to prevent memory leaks must be able to be statically checked, so must be made *explicit* in some way that the checker can understand and validate. Bento encodes this information in type definitions. The ownership contracts are represented as passing data structure ownership or mutable or immutable references, and resource allocation contracts are enforced by objects that provide safe abstractions around other kernel components, freeing the developer from manual resource management. Other techniques could be used to represent these contracts for other methods of static checking, such as including annotations on interfaces.

One potential concern is that these interface requirements will impose some nontrivial performance cost from copies or reduced ability to implement optimizations. While this concern is valid, Rust is generally able to perform similarly to C code, and existing projects have been able to develop Safe Rust operating systems and components that are performance-competitive with Linux. The Bento file system and the RedLeaf [44] and Theseus [13] Rust

operating systems all perform competitively to their Linux counterparts. More research is needed to fully understand the extent to which these proposed interface requirements impose unavoidable performance overhead.

4.4 Functional Correctness

Challenge: Complex Interface Semantics. The Linux kernel often introduces new interfaces with complicated properties and usages for performance reasons. For the monolithic kernel, introducing new functions is easy and adding function calls is cheap, so there's benefit to increasingly complicated interfaces even if the performance gains are corner cases.

The `buffer_head` struct, used to expose disk blocks to file systems through the buffer cache, includes 16 state flags that describe whether the buffer is mapped, dirty, etc. These flags are set independently, resulting in many possible combinations of states. Not all of the combinations are valid, but even determining which are can be complicated. Since these flags control if, when, and how buffers are written to the storage device, they must be set correctly and at the right point in the code to prevent data loss or corruption. If the file system uses an external journal module, such as ext4's jbd2, both need to manage this state and must coordinate with each other and with the buffer cache.

A functionally correct file system interacting with the buffer cache relies on having a correct specification for the `buffer_head` state, but precisely capturing a complex, heavily-coupled implementation entails complex modeling, substantially increasing the proof burden of verification. Some automation techniques also wilt under complex interfaces. Capturing a specification is doubly difficult when two components can modify shared state that affects both components' behavior.

Furthermore, the bottom up approach of specifying individual modules and later composing them into a larger meaningful kernel specification is challenging. We believe that it is almost impossible to know the right interfaces for composed verification before anything is built, therefore, we may need to revisit the interface exported by each module to make composition more feasible. Specifying an end-to-end verified kernel in this manner remains an open question.

Approach: Correctness Guarantees. The ultimate standard in robustness is functional verification: replacing components with ones whose behavior has been shown to meet a precise specification [18, 19, 29–31, 35, 45, 46, 51]. Functional verification excludes all undefined behavior.

Supporting verified modules and their functionally-specified interfaces requires four features: an appropriate interface modeling language, axiomatic models of unverified components, decoupled modules, and ownership specification. Ownership specification is discussed above; we discuss the others next.

Modeling language. A functionally-specified interface unambiguously and abstractly models the behavior of the component behind it. For example, a file system can be modeled as a map from path strings to file content bytes. Similarly, a crash-safe file system can be modeled as a map of path strings to file content bytes that is guaranteed to recover to the last synced version given any crash.

Abstract means exploiting math shorthand to hide implementation details. For example, the directory-rename operation may be modeled as a relation between old and new maps in which every path key with a given prefix is substituted with a new prefix. The specification can freely describe the content of all keys, making it straightforward for users to understand what's expected after each operation. This may sound "expensive", as we are addressing the content of each key in every operation, but since the specification is purely mathematical, this doesn't imply that the implementation is expensive.

Unambiguous means that the model captures all the behavior of the component that client code needs to rely on. Bugs often stem from the author of client code misunderstanding the contract explained in notoriously ambiguous English comments.

Such a model is most easily expressed in a mathematical language with immutable objects (whose meanings don't change as the system evolves) and functions and relations over them. The implementation explains how to "interpret" its efficient, complex, mutable data structure as an instance of the model. Verification shows that each operation performed by the implementation (for example, swinging a pointer in the inode tree to implement rename) is a valid relation between the before- and after- model interpretations.

Verification research has demonstrated that functional languages (immutable data structures, side-effect-free mathematical functions) are better for modeling than repurposed imperative languages (e.g. Dafny's functional subset vs Spec#'s use of C#).

Here we use "functionally correct" to refer to safety properties excluding liveness and performance, but of course performance is necessary for adoption. Verified modules form a foundation for safe performance optimization, especially complex designs that are difficult to implement correctly.

Axiomatic model of unverified code. Since we are concerned with incremental movement, what happens at the boundaries between verified and unverified components? The boundary must provide assumptions (axioms) about the behavior of the unverified module. For example, a verified file system may rely on the behavior of an unverified block I/O layer modeled at the interface. The verified file system will appear buggy if either the block I/O layer is buggy or the model erroneous. Hence, these axioms should be written with minimal assumptions and only cover the basic functionality. In the case of block I/O, the data structure `buffer_head` may be abstracted away, and the

axioms can be defined in terms of bytes. A shim layer is then needed to bridge the communication gap between the verified modules and unverified components. Similarly, this type of shim layer is needed between every incremental boundary.

Note that one can write models of interfaces with unverified code on both sides; this evolution can occur before verification is complete. The process of modeling an interface helps expose poor structuring and tight coupling among ad-hoc modules, misfeatures that need to be refactored before verification can be achieved. While difficult, writing axiomatic models of Linux's complicated interfaces is essential, for if we cannot describe a module's functionality at a high level, then we do not understand its functionality. This is a problem even outside of the context of verification.

Decoupled modules. If a client component employs two subcomponents A and B, it will have models of each. If the behavior of those two subcomponents is coupled, their interfaces cannot be unambiguously described in isolation. That is, calling a function in B may change the interpretation of a data structure retrieved from A, thus the interface to A depends on B. This kind of coupling is common in Linux, where structures are shared across several components for reasons of performance or code accretion. Factoring such data structures into separate per-component data structures makes modeling much more feasible.

Performance-motivated coupling introduces inherent design tension. While the systems verification community has built big multi-component systems [30], we have little experience yet with such multi-component optimizations. Identifying design patterns that balance decoupling and good performance is an open question.

Performant Verified Components. Verification is a compile-time check, and hence in theory presents no inherent limit to code performance. In practice, however, verification often creates a trade-off between programmer ease and performance constraints. For example, some verification frameworks model functional code and extract imperative code [55], such as through a functional language implementation [19]. Even systems that directly model memory-mutating imperative code make assumptions about memory models such as garbage collection [17] or linear ownership [29]. Such assumptions simplify reasoning, but constrain the structure of the code in ways that may exclude faster implementations. Performant OS Rust systems [43, 44] suggest that ownership reasoning is not a barrier to performance. The opposite may even be true: system developers have avoided complex optimizations, such as soft updates [25], that may become feasible with verification. It is worth noting that there has been significant progress on verification techniques and tooling, enabling us to build more and more performant and verified systems. We believe large scale verification has a promising future.

Concurrent Verified Components. With few exceptions [15, 16, 27, 33, 40, 52, 57], few systems verification methodologies reason about shared-memory concurrency. There are simple ways to safely layer concurrent reasoning on top of a single-threaded verification. For example, outsourcing a side-effect-free computation by passing a reference to an immutable data structure is a meta-logically safe extension of a sequential verification result. More progress is needed, however, in proof automation for shared-memory concurrent programs, as exploiting concurrency is essential for good performance. We note that the various steps we have proposed can operate in parallel. By the time clean type and ownership interfaces exist within Linux, the research community may well have made significant progress at some of these seemingly intractable problems.

4.5 Practical Challenges

Rate of Change. The Linux kernel continues to grow at a rate of millions of lines of code per year. Catching up to and then maintaining safety in such a quickly evolving code base remains a challenge. For example, changes must prove that they don't violate existing safety guarantees. For compiler checked properties, such as type safety or ownership safety, this just means that all new code must compile. For verification, existing proofs must be adapted to match new code. Doing this while keeping up with Linux's rate of change requires that local changes to code require similarly local changes to proofs. It is unclear how far we are for knowing how to provide this property or how difficult it will be to engineer systems that do. More experience with verification tools is necessary to ensure that proofs can evolve rapidly with code.

Incentives. Our guiding principle is incremental benefit for incremental work. Society should not have to wait for Linux to be completely verified end-to-end to begin to see benefits in stability and security. In fact, the Linux community has just taken a major step at introducing Rust into the kernel leaf modules [8]. Though there are concerns and integration issues to be solved, the Linux community itself would benefit from a safer Linux. With safe languages, patch reviewers can focus on code functionality, as whole classes of bugs are prevented by safety checks. With verified modules, maintainers can specify the functionality of each interface, and developers are then responsible for the code and proof.

5 Related Work

Linux bug analysis. Chou *et al.* [20] showed that device drivers are the most error-prone components in Linux (up to 2.4). Palix *et al.* [47] looked at Linux up to 2.6 and showed that the total number of new bugs continued to rise. However, the errors per line of code in device drivers was significantly reduced, and the hardware abstraction layer and various file systems became the components with a high fault rate. Our analysis is based on CVE and bug patches in Linux (up to

5.10). Our analysis shows that bugs are being found even in mature modules, such as ext4.

Safe languages or software verification in kernel development. Using safe languages or software verification in kernel development has been extensively studied in the literature. Pilot [48], SPIN [12], Singularity [32], Tock [37], Biscuit [22], and Redleaf [44] are all written in high-level safe languages. SeL4 [35], Hyperkernel [46], and CertiKOS [27] are verified OS kernels. These are clean-slate designs, and it is difficult to directly use their components in Linux. Yggdrasil [51] and FSCQ [18, 19] verify file systems, but they run at userspace using FUSE. Our goal is to complement this line of work by supporting kernel components, developed in safe languages or verified, to run inside Linux. Bento [43] allows developers to write file systems in Rust and load the file systems into the kernel. Our goal is more general: the interface design patterns for modular kernel components written in a safe language or formally verified. Today, Linux already supports loading eBPF, but its expressiveness is limited, and it does not support complex kernel components.

Other mechanisms towards a safe kernel. An alternative method to prevent kernel bugs is to move kernel features into userspace [10, 11, 23, 39, 42]. Kernel components can run as stand-alone services or as part of the application libraries, but this is no magic bullet–bugs that used to be inside the kernel will also move to userspace. Software fault isolation [14, 24, 41, 54] and control flow integrity are also ways to improve kernel safety through runtime checks. This allows memory errors in one kernel component to be isolated from the rest of the kernel. Our goal is both cross and intra-module bug prevention as well as functional verification.

6 Conclusion

We propose an incremental route to a safer Linux through modularization and gradual replacement module by module. The existing Linux design pattern of widely shared data structures with poorly specified module interactions and constraints poses a major barrier to progress. We believe the research community can help push forward this vision by creating alternative, better implementations of major Linux functions, by developing cleaner APIs for kernel functions (such as a new network or virtual memory stack) that support type and/or ownership safety, and attempting partial verification in the context of performance-critical concurrent systems with large codebases.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable comments and helpful feedback. This work is partially supported by the National Science Foundation grant CNS-1856636 AM04 and DGE-1762114. This work was also supported by Google and Huawei.

References

[1] Autotest - Fully automated testing under linux. https://autotest.github.io/.

[2] Coccinelle: A Program Matching and Transformation Tool for Systems Code. https://coccinelle.gitlabpages.inria.fr/website/.

[3] CVE-2020-12351 kernel: net: bluetooth: type confusion while processing AMP packets. https://bugzilla.redhat.com/show_bug.cgi?id=1886521.

[4] Kernel self-test. https://kselftest.wiki.kernel.org/.

[5] KernelCI. https://kernelci.org/.

[6] ktest. https://elinux.org/Ktest.

[7] LKFT - Linaro's Linux Kernel Functional Test framework. https://lkft.linaro.org/.

[8] [PATCH 00/13] [RFC] Rust support. https://lkml.org/lkml/2021/4/14/1023.

[9] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/.

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

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

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

[13] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *OSDI*, 2020.

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

[15] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *OSDI*, 2018.

[16] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying Concurrent, Crash-Safe Systems with Perennial. In *SOSP*, 2019.

[17] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. *CoqPL*, 2020.

[18] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *SOSP*, 2017.

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

[20] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *SOSP*, 2001.

[21] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2.* INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

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

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

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

[25] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 2000.

[26] Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. https://golang.org/.

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*, 2016.

[28] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 517–528, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *OSDI*, 2020.

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP*, 2015.

[31] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI*, 2014.

[32] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 2007.

[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2017.

[34] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[35] 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. SeL4: Formal Verification of an OS Kernel. In *SOSP*, 2009.

[36] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*, 2010.

[37] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *SOSP*, 2017.

[38] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter Sockets can be Fast and Compatible. In *SIGCOMM*, 2019.

[39] Jochen Liedtke. On Microkernel Construction. In *SOSP*, 1995.

[40] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *PLDI*, 2020.

[41] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-principal Modules. In *SOSP*, 2011.

[42] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[43] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *FAST*, 2021.

[44] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *OSDI*, 2020.

[45] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *SOSP*, 2019.

[46] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP*, 2017.

[47] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *ASPLOS*, 2011.

[48] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Commun. ACM*, 1980.

[49] Rust: A language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/.

[50] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 1990.

[51] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *OSDI*, 2016.

[52] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.*, 2020.

[53] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*, 2014.

[54] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993.

[55] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*, 2015.

[56] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[57] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *SOSP*, 2019.