# Kernel extension verification is untenable

Jinghao Jia
University of Illinois
Urbana-Champaign, IL, USA
jinghao7@illinois.edu

Raj Sahu
Virginia Tech
Blacksburg, VA, USA
rjsu26@vt.edu

Adam Oswald
Virginia Tech
Blacksburg, VA, USA
adamoswald@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

Michael V. Le
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
mvle@us.ibm.com

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

## ABSTRACT

The emergence of verified eBPF bytecode is ushering in a new era of safe kernel extensions. In this paper, we argue that eBPF's verifier—the source of its safety guarantees—has become a liability. In addition to the well-known bugs and vulnerabilities stemming from the complexity and ad hoc nature of the in-kernel verifier, we highlight a concerning trend in which escape hatches to unsafe kernel functions (in the form of *helper functions*) are being introduced to bypass verifier-imposed limitations on expressiveness, unfortunately also bypassing its safety guarantees. We propose safe kernel extension frameworks using a balance of not just static but also lightweight runtime techniques. We describe a design centered around kernel extensions in safe Rust that will eliminate the need of the in-kernel verifier, improve expressiveness, allow for reduced escape hatches, and ultimately improve the safety of kernel extensions.

## CCS CONCEPTS

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

## KEYWORDS

Operating system, kernel extension, eBPF, verification, safety

**ACM Reference Format:**
Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel extension verification is untenable. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595892


*HotOS '23, June 22–24, 2023, Providence, RI, USA*

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595892

## 1 INTRODUCTION

The emergence of a popular safe kernel extension framework in the form of *eBPF* in Linux has ignited an industry around system-level capabilities from tracing and observability [21] all the way to security [26], networking [23], storage [20, 52], and consensus [53]. Core to its value proposition is an unprecedented promise of safety. To this end, eBPF programs are compiled to a restricted bytecode, upon which the kernel performs *verification*: a form of symbolic execution to examine all possible program paths and guarantee properties, including memory safety, freedom from crashes, proper resource acquisition and release, and termination.

Unfortunately, the current in-kernel eBPF verification approach does not live up to its promise of safety. An increasing tide of concerns is rising in the community questioning the correctness of the in-kernel verifier, which continues to grow in complexity. Kernel bugs introduced by the verifier, as well as exploits leveraging unsafe extensions that pass the verifier but violate safety properties, are constantly reported (see §2.1). Efforts are underway to improve the eBPF verifier through fuzzing [41], verifying the verifier [11], or rewriting the verifier with proof-carrying code [39].

However, even if the verifier were flawless, we observe that verified code makes up only a small portion of an extension program. In eBPF, verified code interacts with a growing set of potentially complex and unverified *helper functions*, which serve as "escape hatches" to make up for the severe limitations on program expressiveness required for verification (see Figure 1). In fact, by using helper functions, verified "safe" eBPF programs can violate all of the verification guarantees mentioned above. So, although extension writers pay a heavy cost to program within the constraints of the in-kernel verifier, they do not receive the promised safety guarantees.

We take the position that the current myopic approach to safe kernel extensions that relies solely on static bytecode

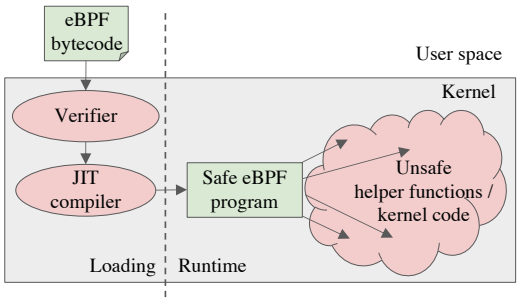


**Figure 1: Overview of eBPF and helper functions**

verification is untenable. Instead, we argue for a broader approach towards safe kernel extensions beyond static bytecode verification. Inspired by the use of language safety in OS kernels past [10] and present [12, 37], our key insight is that, through a balance of language safety, runtime protection, and separation of concerns between checking safety and performing kernel duties, the extension framework can be more expressive with similar safety. That increased expressiveness reduces the need for dangerous helper functions, resulting in better, more practical guarantees.

We propose kernel extensions to be written using the Rust programming language, because its approach to safety—including but not limited to memory safety, undefined behavior, and resource ownership—has been explored in other OS contexts [9, 12, 31, 37] and embraced by Linux [8]. Rather than attempting to check safety properties in the kernel, we allow a trusted userspace Rust toolchain to sign extensions and leverage secure key bootstrap mechanisms to validate signatures at load time. Finally, we suggest lightweight runtime mechanisms that complement Rust to achieve properties, such as program termination, that are not easy to do statically without severely impacting expressiveness.

We believe that moving to safe, expressive language-based extensions is a key step to continuing the growth of the industry and the emergence of more complex use cases around safe kernel extensions. Furthermore, as a new entry point to implement kernel functionality in a safe language, we believe safe Rust extensions will be an important tool to answer the call to arms towards safe and practical OS kernels [31].

## 2 VERIFICATION IS NO GUARANTEE

Despite the excitement and promise surrounding eBPF verification, kernel extensions have not achieved the safety properties that one would expect. As a result, the kernel community is cautious about eBPF, going so far as to reject use cases that would allow unprivileged users to load (verified) kernel extensions [22]. Here we provide more details about the known issues with the verifier, identify new challenges to verifier guarantees caused by helper functions, and make a case for a move away from the verifier.

### 2.1 Verification is not easy

It is known that the current eBPF verifier in Linux is buggy and brittle, due to its increasing complexity, constant changes, as well as the challenges of sound and complete static analysis [19, 33, 39, 50]. Here we highlight the growth in the complexity and bugs of the verifier, as well as its costs.

**The verifier complexity is growing.** As shown in Figure 2, the eBPF verifier has been growing in size to support new checks for features since it was introduced in 2014. For example, with the introduction of the `bpf_spin_lock` helper, the verifier grew to check that an eBPF program only holds one lock at a time and releases the lock before termination in any execution [48]. To support BPF-to-BPF calls, 500 lines of C code were added to the verifier [45]. Meanwhile, the verifier has been under constant optimization and refactoring to reduce verification time and memory consumption. The multitude of new verifier features being actively developed (e.g., [18, 49]) indicates that eBPF has not yet achieved adequate expressiveness. We do not expect the growth to subside in the near future.

**The verifier is buggy.** The ever-increasing complexity leads to new bugs being continuously introduced. Table 1 shows that at least 22 bugs were discovered in the eBPF verifier in the past two years. These bugs open up two types of exploits.

First, a buggy verifier could accept unsafe, malicious eBPF programs, allowing attacks like arbitrary read and write [2, 4, 5], kernel pointer leak [3, 13–15, 32], and privilege escalation [2, 4]. For example, in a recent bug documented by CVE-2022-23222 [4], missing validation on pointer values allows unprivileged users to perform illegal pointer arithmetic, causing arbitrary read and write capabilities on kernel memory, and eventually privilege escalation. Second, the verifier itself can be vulnerable and exploited by unsafe, malicious eBPF programs. For example, a recent commit [54] fixed a use-after-free bug in the loop-inlining code of the verifier.

In addition, even a perfectly coded verifier cannot prevent malicious eBPF programs from exploiting bugs in downstream components of the eBPF ecosystem such as the JIT compiler [38]. For example, a recent bug in the JIT compiler [1] allows malicious eBPF code that successfully passes the verifier to hijack the kernel control flow.

**Verification is expensive.** Verification costs both human time and machine time. It is known that the verifier frequently reports false positives that unnecessarily force developers to heavily massage correct eBPF code to pass the verifier [19, 39, 50]. A more fundamental problem is the limited scalability of the verifier. Since the verifier needs to evaluate all possible execution paths, it has to limit the eBPF program size and complexity to complete the verification in time. To satisfy these verifier limits, developers need to find ways to break their program into small pieces when

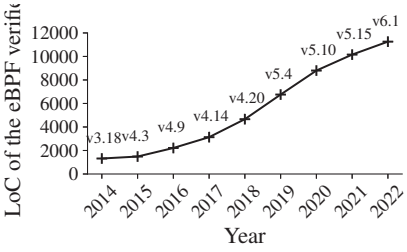


**Figure 2: Lines of code of the eBPF verifier by kernel over time**

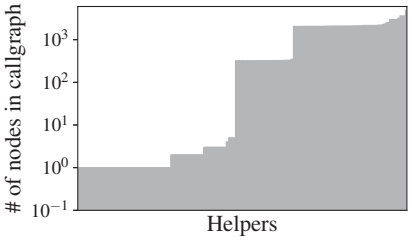


**Figure 3: Call-graph complexity of each eBPF helper**

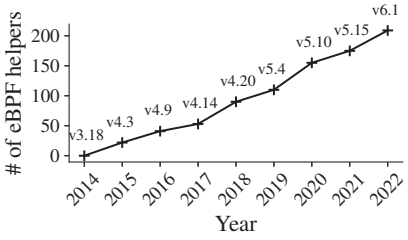


**Figure 4: The number of helper functions by kernel versions and by year**

| Vulnerabilities/Bugs | Total | Helper | Verifier |
|---|---|---|---|
| Arbitrary read/write | 3 | 1 | 2 |
| Deadlock/Hang | 2 | 1 | 1 |
| Integer overflow/underflow | 2 | 2 | 0 |
| Kernel pointer leak | 5 | 0 | 5 |
| Memory leak | 2 | 0 | 2 |
| Null-pointer dereference | 7 | 6 | 1 |
| Out-of-bound access | 7 | 1 | 6 |
| Reference count leak | 1 | 1 | 0 |
| Use-after-free | 2 | 1 | 1 |
| Misc | 9 | 5 | 4 |
| Total | 40 | 18 | 22 |

**Table 1: Bug statistics in eBPF helper functions and verifier in years of 2021 and 2022.** Instead of searching CVEs (which are not embraced by the Linux community [28]), we searched commit logs for security-related bug fixes and confirmed them manually.

writing large, complex programs [20]. The result is reduced programmability and increased performance overhead [29].

## 2.2 Verified code needs help(ers)

Even with a correctly implemented verifier, the promise of safety is still hard to achieve, because verified code interfaces with unsafe kernel code in the form of *helper functions*. As shown in Figure 1, helper functions (helpers) are normal, unverified kernel functions that generally provide read / write access to various kernel data structures (e.g., a socket buffer). Helpers offer escape hatches for eBPF programs to become more useful, as complex logic or out-of-program memory accesses may not be expressible in eBPF or verifiable by the in-kernel verifier. On the other hand, helpers provide a direct mechanism for verified code to misbehave.

**Helper functions are complex.** It is generally accepted that complex code tends to have more bugs than simple code. To measure the complexity of helper functions and a first indication of their potential danger, we statically analyzed the Linux kernel version 5.18 to compute the call graph of each helper functions. Figure 3 shows the number of unique nodes in the call graph of each of the 249 helper functions in Linux-5.18.[1] As shown in the figure, helper functions vary in their complexity. For example, `bpf_get_current_pid_tgid`, which retrieves the PID and TGID of the current task, calls no other kernel functions. On the other hand, `bpf_sys_bpf`, which allows eBPF programs to invoke a subset of the `bpf` system call, has 4845 nodes in its callgraph. Specifically, 52.2% of the helper functions call 30+ other kernel functions and 34.5% call 500+ other functions. Bugs and vulnerabilities in the helper function implementations are a natural consequence of their complexity, which, as described below, can be exploited by unsafe or malicious eBPF programs.

**Helper functions are growing.** The main motivation to introduce new helper functions is to increase the expressiveness and utility of eBPF programs. As researchers and practitioners invent new use cases of safe kernel extensions, rather than implementing these new use cases in eBPF and passing them through the verifier, they are resorting to introducing new helper functions. Figure 4 shows the growth in helpers as a function of time. Roughly 50 helper functions are added every two years. In addition to these helper functions that are specifically developed and exposed for eBPF programs, developers also have introduced new ways for exposing *existing* internal kernel functions for eBPF programs to use [16]. Since these internal kernel functions were not written with eBPF usage in mind, it is even more likely that their use by eBPF programs will result in safety violations. With this trend, in the next decade, the helper function interface will be as wide as (or wider than) the system call interface, providing many opportunities for verified code to trigger unexpected behavior.

**Helper functions can be lawbreakers.** With more new helpers being introduced, bugs and vulnerabilities are constantly being discovered in various helper functions. As shown in Table 1, at least 18 security-related bugs have been found and fixed in the Linux kernel in the past two years. These results show that helper functions are far from being safe and can easily violate properties assumed by the verifier.

[1] Note that these numbers are lower bounds—our static analysis did not account for function pointers.

To concretely show the dangers of helper functions today, we examine two different properties that are guaranteed by the verifier—safety and termination.

- **Safety.** The verifier ensures that eBPF code cannot access memory outside the program, including trying to dereference a NULL pointer. However, through a helper function, we wrote eBPF programs that crash the kernel. Specifically, we discovered a bug in the helper `bpf_sys_bpf` and constructed an eBPF program to call the helper with a union pointer argument containing a NULL pointer field. Since the verifier does not perform deep argument inspection, we achieved a kernel crash by dereferencing the NULL pointer inside the union. We reported this bug, which soon was determined to be exploitable (allowing an arbitrary kernel read) and assigned a CVE [5].
- **Termination.** The eBPF verifier is supposed to guarantee termination to prevent kernel lockups caused by buggy or malicious eBPF programs. However, we can easily craft an eBPF program that runs for practically infinite time while holding the RCU read lock, causing RCU stalls. Our crafted eBPF code uses nested calls to the `bpf_loop` helper to perform random reads and writes on an eBPF map object. It gives us linear control over total runtime; while we have run it continuously for 800 seconds (more than enough to observe RCU stalls), we calculate that with more nested loops and eBPF tail calls [44], we can craft a program that will run for millions of years.

## 2.3 The eBPF verifier needs to retire

Taking a step back, there are two reasons the current eBPF verification approach is inadequate:

- Static bytecode verification has both soundness and completeness issues, and is fundamentally hard to scale, which inevitably admits unsafe code.
- Unreasonable constraints on extension expressiveness result in the introduction of unsafe escape hatches in the form of helper functions.

To date, the community has focused on the first issue, primarily by improving the verifier implementation. PREVAIL uses abstract interpretation to implement a userspace verifier [19]. Fuzzing and formal verification have been proposed to improve both the existing verifier and the JIT compiler [11, 38, 39, 41, 50, 51]. Decoupling the burden of proof from the kernel is being explored with proof-carrying code [39].

Unfortunately, to the best of our knowledge, the issue of unsafe helper functions has been overlooked; we expect that even with the aforementioned advances in verification, helper functions will continue to undermine safety.

# 3 BEYOND VERIFICATION

Instead of continuing to go down the path of static bytecode verification—which is ineffective—we make the following suggestions for a new approach to safe kernel extensions:

- **The extension language should be more expressive.** A more expressive safe language can eliminate the need for some helpers and simplify others.
- **Static code analysis should be decoupled from the kernel.** Leveraging the broader (userspace) communities working on type checkers and formal software verification reduces bugs stemming from ad hoc implementation.
- **Static analyses and runtime mechanisms should work together.** Implementing properties that are easy and efficient to enforce in the runtime reduces the burden on analysis and/or verification (and its complexity).

In the rest of this section, we describe a potential architecture for safe kernel extensions that does not require overly restrictive verification and thereby avoids its pitfalls.

## 3.1 A Rust-based approach

We propose that instead of relying entirely on in-kernel static bytecode verification using execution simulation, safe kernel extensions should rely on a combination of language safety and lightweight runtime mechanisms. Figure 5 gives an overview of the proposed kernel extension framework.

**Rust for safety properties.** Rust is emerging as a popular systems programming language—even for OS kernels [9, 12, 31, 37]—due to its lightweight abstractions, effective elimination of undefined behavior (e.g. memory errors or integer errors), and unique memory ownership model. By restricting user-implemented extension programs to only use safe Rust (i.e., no `unsafe` blocks), the Rust compiler takes the role of the verifier to ensure the code is safe to run. We envision a trusted "kernel crate" that provides the interface between the safe Rust of the extension program and the kernel.

In addition to memory and integer safety, Rust can also enforce properties relating to safe resource acquisition and release. For example, in eBPF, the verifier currently checks for the proper release of the resources acquired by the program via helper functions (e.g., the reference count obtained from the `bpf_sk_lookup_tcp` helper and the spin lock acquired from the `bpf_spin_lock` helper function), rejecting programs that can possibly leave dangling resources. In Rust, the resource-acquisition-is-initialization (RAII) pattern [7] can be used to create an abstraction around kernel resources that user extension code must use. When the object goes out-of-scope, the resource is automatically released in the destructor, guaranteeing its proper release.

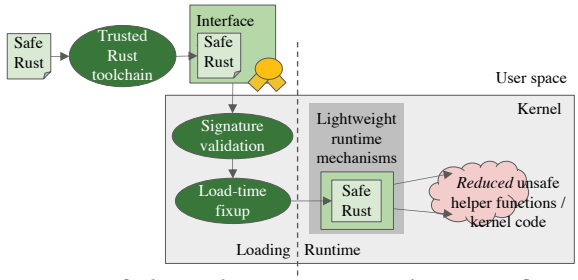


**Figure 5: Safe kernel extensions without verification**

**Decoupling static code analysis.** Rather than analyzing the code to ensure safety in a one-off implementation inside the kernel at load time, we leverage the full Rust community, toolchain, and the many ongoing Rust verification projects [40] to perform safety checks. By piggybacking on kernel support for signed kernel modules (and even signed eBPF programs [43]), our architecture involves a trusted compiler that checks and signs an extension program (see Figure 5). At load time, the kernel checks the signature to ensure safety. The kernel may need to perform some amount of load-time fixup on the program to resolve helper function addresses and other relocations, but it does not incur the burden (and complexity) of checking safety properties.

**Runtime protection.** As a general-purpose programming language, even programs using the safe subset of Rust can exhibit undesirable behavior, including infinite loops or deadlocks. While we rely on the Rust language for memory isolation and prevention of undefined behavior, we use runtime mechanisms like watchdog timers, signals, and stack protection to terminate the program rather than violate safety. Related work has also explored the use of hardware protections at runtime, including lightweight page protection keys to augment language safety [27, 30, 33].

A key challenge raised by runtime mechanisms is how to perform safe termination of an extension program. It is critical that any allocated kernel resources (e.g., reference counts) are released upon termination for any reason (watchdog timeout, Rust's own panics). While in userspace Rust uses an ABI-based stack unwinding mechanism (e.g., `llvm-libunwind`) to handle exceptions and to perform cleanup operations, such a method is not desirable for kernel extensions:

- Failures during unwinding, which are permissible in userspace, cannot be tolerated in kernel space, as incomplete cleanup means leaking kernel resources.
- ABI-based unwinding typically requires dynamic allocation, which creates challenges for extensions in interrupt contexts, in which an allocator may not be available [17].
- Unwinding generally executes destructors for all existing objects on the stack, but executing untrusted, user-defined destructors (via the `Drop` trait in Rust) is not safe.

| Safety properties | Enforcement |
|---|---|
| No arbitrary memory access | Language safety |
| No arbitrary control-flow transfer | Language safety |
| Type safety | Language safety |
| Safe resource management | Runtime protection |
| Termination | Runtime protection |
| Stack protection | Runtime protection |

**Table 2: Safety properties and the enforcement mechanisms of the proposed extension framework**

In our framework, light-weight mechanisms can be effective for cleaning up kernel resources. We can record allocated kernel resources and their destructors on-the-fly during program execution. When termination is needed, the destructors of allocated resources are invoked to release the resources. Since only the trusted kernel crate that interfaces with the kernel resources is responsible for implementing the aforementioned destructors, all the cleanup code is trusted and guaranteed not to fail. To deal with dynamic allocation of the unwind context, we envision using a memory-pool-based allocation mechanism or avoiding dynamic allocation altogether with a dedicated per-CPU region for storage.

**Safety properties.** Table 2 summarizes the major safety properties normally enforced by the verifier that can instead be enforced by the proposed kernel extension framework through language safety and runtime protection. Unlike eBPF, they are achieved without restrictions on loop and program size. We discuss other verified properties in §4.

## 3.2 Safety without escape hatches

The fact that Rust is a high-level Turing-complete language provides the advantage of better programmability in contrast to the restricted subset of C in the current eBPF programming model. In this section, we discuss classes of helper functions that can either be completely eliminated by leveraging the increased expressiveness of Rust, or be simplified and made safer by rewriting some aspects of the functions in safe Rust.

First, the helpers introduced to compensate for the lack of expressiveness of the eBPF language can be retired. We use `bpf_loop`, `bpf_strtol` and `bpf_strncmp` as three representative examples: (1) `bpf_strtol` can be replaced by the built-in `core::str::parse` in Rust, (2) `bpf_strncmp` can be implemented entirely in safe Rust, without the need to call into unsafe C code in the kernel, and (3) `bpf_loop` can be directly removed given that it merely provides a loop mechanism. According to a preliminary study [33], 16 of the helper functions fall in this category and may be retired.

Second, many helpers for interfacing with kernel objects and procedures cannot be entirely removed but can be greatly simplified, with safe Rust replacing error-prone C code. Table 1 shows two bugs that cause reference count leaks in two

helper functions, `bpf_get_task_stack` and `bpf_sk_lookup` [34, 35]. With Rust, such vulnerabilities can be prevented using the ownership system. Using the RAII pattern, a Rust abstraction of the referenced object can be implemented to hold the reference for its lifetime, effectively releasing the reference count when it goes out-of-scope. Another example is integer arithmetic. Since Rust prohibits undefined behavior stemming from integer errors (e.g. overflow, as the bug in array map helpers [36]) via runtime checks, integer arithmetic can be moved from helpers into safe Rust. When a program invokes such a helper using the interface provided by the kernel crate, integer operations are performed before the Rust code calls into the unsafe kernel implementation, thereby preventing integer errors in unsafe code.

Lastly, helpers can be made safer by implementing a safe interface on top of the unsafe code. The interface can provide mitigations for vulnerabilities that manifest from unsanitized input to helpers that the verifier fails to check. In Table 1, `bpf_task_storage_get` has a null pointer dereferencing bug when the helper receives a null `task_struct` pointer [42]. The helper can be wrapped in Rust with the `task_struct` pointer argument being a reference type—the Rust compiler will ensure the program always has to borrow the reference from a valid object, effectively preventing such vulnerabilities. The same interface can be implemented for `bpf_sys_bpf`, mitigating the vulnerability discussed in §2.2.

We believe that refactoring the cubersome, complex helper interface into a simple, well-specified interface can largely resolve the battle between safety and expressiveness, as explored in other contexts [24].

## 4 OPEN QUESTIONS AND DISCUSSION

**Further verification guarantees.** Most verifier guarantees are achievable either through Rust or a runtime mechanism. Recently, the verifier has included logic to reject / sanitize programs that contain gadgets that train branch predictors or similar to facilitate transient execution side channel attacks [46, 47]. While similar strategies could be applied on the Rust-level or binary level, there is a fundamental tradeoff between increased expressiveness for extensions (which helps reduce unsafe helper functions) and the availability of program information to statically provide guarantees. We believe that safe Rust provides a good balance given the current state of the art. Furthermore, efforts to bridge the gap by providing formal guarantees about Rust are ongoing [6].

**Dynamic memory allocation.** The existing eBPF subsystem does not support dynamic memory allocation in eBPF programs, which makes them easier to verify [19]. With the proposed approach in Rust, it is possible to integrate a memory allocation framework for the extension programs. Such a framework can use a pre-allocated memory pool implementation [17], given that extension programs often run in a non-sleepable context (e.g., from kernel interrupts). Dynamic allocation greatly enhances the programmability of kernel extensions, allowing them to support more complicated use cases. Certainly, dynamic memory management brings challenges to ensure safety. Even though the user programming interface can be implemented in safe Rust, as the case of the current Rust standard library, low-level memory management code usually has to be written in unsafe Rust.

**Protection from unsafe code.** The concept of a single address space system, where language safety provides isolation between processes [25], thereby eliminating the need for expensive hardware context switches, has recently been revisited in the context of Rust [12, 37]. For kernel extensions, however, the threat of an errant write from unsafe code into code or data belonging to the safe extension is unavoidable. Unsafe code is used to invoke helper functions or implement low-level systems routines in the kernel crate; in fact, the majority of the kernel itself is unsafe. Lightweight hardware-supported memory protection [27, 30, 33] seem a promising technique to protect safe code from unsafe code, but raise an interesting question: if we must resort to hardware protection mechanisms, is language safety or verification still necessary to protect the kernel and extensions from one another? Even if not, the use of safe languages is one step towards the future possibility of an entirely safe kernel that can realize the benefits of prior single address space systems.

## 5 CONCLUSION

The vast potential of safe kernel extensions is being stunted by limitations of the in-kernel eBPF verifier. Moving away from the verifier will lead to a safer, more expressive kernel extension framework. The key is a balance of static analysis techniques with lightweight dynamic mechanisms. The Rust ecosystem provides the ideal properties for such balance, while also being well positioned to leverage improvements to program verification. At the same time, by separating concerns, a Rust-based extension framework can utilize increasingly lightweight hardware features. Finally, as Rust extensions enable more and more kernel code (e.g., helper functions) to migrate to safe Rust, new opportunities arise, not only for kernel extension use cases, but also to re-implement critical kernel subsystems and ultimately lead to a safe, trustworthy OS kernel.

## ACKNOWLEDGEMENT

We thank Md Sayeedul Islam and Wentao Zhang for early participation in the project. Williams' group is supported in part by NSF grant CNS-2236966. Xu's group is supported in part by NSF grant CNS-1956007 and an Intel gift.

## REFERENCES


[1] CVE-2021-29154. https://nvd.nist.gov/vuln/detail/CVE-2021-29154.

[2] CVE-2021-31440. https://nvd.nist.gov/vuln/detail/CVE-2021-31440.

[3] CVE-2021-45402. https://nvd.nist.gov/vuln/detail/CVE-2021-45402.

[4] CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[5] CVE-2022-2785. https://nvd.nist.gov/vuln/detail/CVE-2022-2785.

[6] ERC Project "RustBelt". https://plv.mpi-sws.org/rustbelt/.

[7] RAII - Rust By Example. https://doc.rust-lang.org/rust-by-example/scope/raii.html.

[8] Rust for Linux - GitHub. https://github.com/Rust-for-Linux.

[9] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)* (May 2017).

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

[11] Bhat, S., and Shacham, H. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf, May 2022.

[12] Boos, K., Liyanage, N., Ijaz, R., and Zhong, L. Theseus: An Experiment in Operating System Structure and State Management. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[13] Borkmann, D. bpf: Fix kernel address leakage in atomic cmpxchg's r0 aux reg. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a82fe085f344ef20b452cd5f481010ff96b5c4cd, Dec. 2021.

[14] Borkmann, D. bpf: Fix kernel address leakage in atomic fetch. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=7d3baf0afa3aa9102d6a521a8e4c41888bb79882, Dec. 2021.

[15] Borkmann, D. bpf: Fix insufficient bounds propagation from adjust_scalar_min_max_vals. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3844d153a41adea718202c10ae91dc96b37453b5, July 2022.

[16] Corbet, J. Calling kernel functions from BPF. https://lwn.net/Articles/856005/, May 2021.

[17] Corbet, J. A BPF-specific memory allocator. https://lwn.net/Articles/899274/, June 2022.

[18] Corbet, J. The BPF panic function. https://lwn.net/Articles/901284/, July 2022.

[19] Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J. A., Rinetzky, N., Ryzhyk, L., and Sagiv, M. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)* (June 2019).

[20] Ghigoff, Y., Sopena, J., Lazri, K., Blin, A., and Muller, G. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (Apr. 2021).

[21] Gregg, B. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html.

[22] Gupta, P. bpf: Disallow unprivileged bpf by default. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8a03e56b253e9691c90bc52ca199323d71b96204, Oct. 2021.

[23] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)* (Dec. 2018).

[24] Howell, J., Parno, B., and Douceur, J. R. Embassies: Radically Refactoring the Web. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)* (Apr. 2013).

[25] Hunt, G. C., Larus, J. R., Abadi, M., Aiken, M., Barham, P., Fahndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B. An Overview of the Singularity Project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[26] Jia, J., Zhu, Y., Williams, D., Arcangeli, A., Canella, C., Franke, H., Feldman-Fitzthum, T., Skarlatos, D., Gruss, D., and Xu, T. Programmable System Call Security with eBPF. *arXiv:2302.10366* (Feb. 2023).

[27] Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., and Franz, M. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)* (Apr. 2022).

[28] Kroah-Hartman, G. Cves are dead, long live the cve! https://kernel-recipes.org/en/2019/talks/cves-are-dead-long-live-the-cve/, Sept. 2019.

[29] Kuo, H.-C., Chen, K.-H., Lu, Y., Williams, D., Mohan, S., and Xu, T. Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)* (Apr. 2022).

[30] Li, H., Gu, J., Xia, Y., Zang, B., and Chen, H. Memory Isolation Mechanism of eBPF Based on PKS Hardware Feature. In *Journal of Software (China)* (2022), pp. 1–18.

[31] Li, J., Miller, S., Zhuo, D., Chen, A., Howell, J., and Anderson, T. An Incremental Path towards a Safer OS Kernel. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).

[32] Li, Y. bpf: Fix wrong reg type conversion in release_reference(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f1db20814af532f85e091231223e5e4818e8464b, Nov. 2022.

[33] Lu, H., Wang, S., Wu, Y., He, W., and Zhang, F. MOAT: Towards Safe BPF Kernel Extension. *arXiv:2301.13421* (Mar. 2023).

[34] Marchevsky, D. bpf: Refcount task stack in bpf_get_task_stack. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=06ab134ce8ecfa5a69e850f88f81c8a4c3fa91df, Mar. 2021.

[35] Maxwell, J. bpf: Fix request_sock leak in sk lookup helpers. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3046a827316c0e55fc563b4fb78c93b9ca5c7c37, June 2022.

[36] Nakryiko, A. bpf: fix potential 32-bit overflow when accessing ARRAY map element. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=87ac0d600943994444e24382a87aa19acc4cd3d4, July 2022.

[37] Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G., and Burtsev, A. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (Nov. 2020).

[38] Nelson, L., Van Geffen, J., Torlak, E., and Wang, X. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[39] Nelson, L., Wang, X., and Torlak, E. A proof-carrying approach to building correct and flexible in-kernel verifiers. In *Linux Plumbers Conference* (Sept. 2021).

[40] Reid, A. Automatic Rust verification tools (2021). https://alastairreid.github.io/automatic-rust-verification-tools-2021/, June 2021.

[41] Scannell, S. Fuzzing for ebpf jit bugs in the linux kernel. https://scannell.io/posts/ebpf-fuzzing/, 2021.

[42] Singh, K. bpf: Local storage helpers should check nullness of owner ptr passed. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1a9c72ad4c26821e215a396167c14959cf24a7f1, Jan. 2021.

[43] Singh, K. BPF Signing and IMA integration. https://lpc.events/event/16/contributions/1357/, Sept. 2022.

[44] Starovoitov, A. bpf: allow bpf programs to tail-call other bpf programs. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=04fd61ab36ec065e194ab5e74ae34a5240d992bb, May 2015.

[45] Starovoitov, A. bpf: introduce function calls (verification). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f4d7e40a5b7157e1329c3c5b10f60d8289fc2941, Dec. 2017.

[46] Starovoitov, A. bpf: Prevent memory disambiguation attack. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=af86ca4e3088fe5eacf2f7e58c01fa68ca067672, May 2018.

[47] Starovoitov, A. bpf: prevent out-of-bounds speculation. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=b2157399cc9898260d6031c5bfe45fe137c1fbe7, Jan. 2018.

[48] Starovoitov, A. bpf: introduce bpf_spin_lock. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=d83525ca62cf8ebe3271d14c36fb900c294274a2, Jan. 2019.

[49] Vernet, D. Long-lived kernel pointers in BPF. https://lwn.net/Articles/900749/, July 2022.

[50] Vishwanathan, H., Shachnai, M., Narayana, S., and Nagarakatte, S. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *Proceedings of the 2022 IEEE Symposium on Code Generation and Optimization (CGO'22)* (Apr. 2022).

[51] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock., Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[52] Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., and Cidon, A. XRP: In-Kernel storage functions with eBPF. In *Proceedings of 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[53] Zhou, Y., Wang, Z., Dharanipragada, S., and Yu, M. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)* (Apr. 2023).

[54] Zingerman, E. bpf: Fix for use-after-free bug in inline_bpf_loop. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=fb4e3b33e3e7f13befdf9ee232e34818c6cc5fb9, June 2022.