KLean: Extending Operating System Kernels with Lean

Di Jin
Brown University
Providence, Rhode Island, USA
di_jin@brown.edu

Ethan Lavi
Brown University
Providence, Rhode Island, USA
ethan_lavi@brown.edu

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

Robert Y. Lewis
Brown University
Providence, Rhode Island, USA
robert_lewis@brown.edu

Nikos Vasilakis
Brown University
Providence, Rhode Island, USA
nikos@vasilak.is

Abstract

Safe kernel extension is an extremely successful feature in OS kernels with a plethora of interesting applications. It provides significant performance benefits by avoiding context switching and data copying, without compromising the kernel's integrity due to its verifiable safety. The most mature existing approach, namely BPF, verifies extension safety using sound abstract interpretation techniques with best effort precision. Such design not only increases the kernel maintenance burden due to its complexity, but also restricts extension expressiveness due to its approximations. The core of the problem, we argue, is the BPF verifier's dual mandate of precision and soundness in its safety analysis.

We propose KLean: a Lean-based kernel extension framework that decouples safety reasoning, offloading the safety proving responsibility to user space while only implementing safety specification and checking in the kernel. KLean's design significantly reduces the burden for kernel maintenance, while giving users the full expressiveness power of Lean. We envision KLean to enable significant advancement in safe kernel extension applications.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Formal methods**; *Software safety*.

Keywords: BPF, Verification, OS Kernels, Kernel extensions, Formal methods

ACM Reference Format:
Di Jin, Ethan Lavi, Jinghao Jia, Robert Y. Lewis, and Nikos Vasilakis. 2025. KLean: Extending Operating System Kernels with Lean. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3764860.3768336

1 Introduction

Verifiably safe OS kernel extensions are madly in need, as indicated by the increasing popularity of BPF in areas like tracing [49], networking [44, 77, 90, 98], storage [96], memory management [27, 95], and scheduling [48, 53]. The vibrant BPF mailing list has long been filled with questions and feature requests.

Core to BPF's design is its safety promise through static verification. When a BPF program is loaded into the kernel, the in-kernel BPF verifier performs abstract interpretation on all possible execution paths (with search pruning) and verifies a set of safety properties such as memory safety and termination. However, the BPF verifier has significantly increased in complexity since its original adoption in Linux, leading to a long list of vulnerabilities [10–13]. Moreover, the restrictions imposed by the verifier have become constraining factors for BPF program expressiveness [39, 44, 50]. We argue that the current design requires the BPF verifier to achieve high precision in a sound static analysis algorithm, which is fundamentally difficult.

To address this issue, we propose KLean: a Lean framework for implementing safe kernel extensions. Lean enables the decoupling of the problem: it ensures safety verification via its core type checker, allows expressive safety specification through its dependent type system, and most importantly, enables offloading of the safety proving responsibility completely to user space. Other Lean features also make it suitable for safe kernel extension: (1) built-in, extensible proof support and tactics that simplify proof writing, (2) battery-included libraries and built-in optimization mechanisms for efficient run-time performance, (3) powerful meta-programming infrastructure that enables domain-specific language and proof automation.

We foresee interesting challenges such as improving the performance *vs.* TCB-size tradeoff, sound and precise resource consumption estimation, *etc.* KLean's design will expand the domain of safe OS kernel extension, and open up unprecedented opportunities for system research, including efficient system call virtualization, complete datapath delegation, KLean-aware compiler optimizations, and so much more.

2 Background

2.1 BPF and Safe OS Kernel Extension

BPF [69] started as a simple domain-specific language to filter packets for network monitoring applications, saving CPU cycles on processing and delivering unnecessary packets to the user process. Since then, BPF has evolved into a general-purpose safe kernel extension mechanism, with a wide range of applications.

BPF programs are typically written in high-level languages such as C and Rust, then compiled into BPF bytecode. BPF bytecode is defined similar to a RISC instruction set, enabling simpler JIT compilation [31]. Toolchains like BCC [49], LLVM [64], and libbpf [5] help transform high-level languages to BPF bytecode, simplifying the development.

The safety of a BPF program is ensured by the BPF verifier, which verifies safety properties — such as memory safety, termination, type safety, and invariants related to locks and memory allocation — on BPF bytecode. BPF programs that pass the verifier are accepted by the kernel, and can then be JIT-ed and attached to BPF hooks. The BPF hooks refer to the kernel's definition of how a BPF is registered and invoked at specific extension points. Later, when execution reaches the hook (*e.g.,* a packet arrives), the BPF program is triggered and executed, with a domain-specific data structure (dubbed the context object) as the argument.

The kernel provides two other functionalities to BPF's runtime. *BPF maps* are a set of pre-defined data structures (*e.g.,* array, hash map) that store persistent state across BPF invocations and communicate with the user space. *BPF helpers*[1] are native functions that can be invoked from BPF programs to interact with a subset of kernel states, as well as implementing features that cannot be easily verified, such as loops and string manipulations.

2.2 Calculus of Inductive Constructions and Lean

The *Calculus of Inductive Constructions* (CIC), a family of type theories allowing for dependent types, is recognized for its use in verified programming. These type theories are expressive enough to encode arbitrary mathematical propositions as types. Languages based on the CIC, such as Rocq [91], Lean [72], and Agda [26], permit users to write programs that are *correct by construction*: they can write arbitrarily detailed specifications of the observable behaviors of their programs and prove that these specifications are met.

Unlike many frameworks for program verification, such languages fully separate proof *search* from proof *checking*. Users are expected to construct a proof term, typically in interaction with untrusted language components and automation; this proof term is then checked by a small trusted core. While determining whether an arbitrary specification holds is an undecidable problem, checking whether a given proof is correct is identical to type-checking a program and is decidable and efficient.

Languages with this foundation and architecture are often classified as *proof assistants*, emphasizing proof-checking over executable code. Lean 4 has recently gained recognition as a full-fledged functional programming language, featuring a mature standard library, performant runtime, compiler producing reliable and optimized code, and significant self-reflection capabilities [72]. Its "local imperativity" features [92] allow programmers to write code with attention to memory management while remaining in a functional, verifiable fragment of the language. Lean's core library and mathematical library [68] contain powerful tactics to assist users in constructing proof terms; with its metaprogramming framework [40], users can easily add their own domain-specific proof automation.

3 Motivation

The entire BPF subsystem depends on the BPF verifier: BPF's safety hinges on the verifier's soundness, and BPF's expressiveness on the verifier's precision. However, some safety properties are hard to analyze with a typical type-inference-style algorithm. As a result, BPF struggles from not only verifier soundness issues due to its complex algorithm, but also limited program expressiveness. In this section, we will discuss impact of BPF verification on kernel code complexity and maintenance burden, as well as the expressiveness restriction on BPF programs.

3.1 Verifier Complexity

Memory safety has been a core guarantee since BPF's inception. The verifier performs path-sensitive abstract interpretation on the value range domain for scalars, used to determine whether a pointer offset is within bound of the object. To support safety reasoning in specific code patterns, a constant stream of precision improvement changes was added over time. These include control-flow state refinement [82], ALU precision improvement [81], state pruning and merging [88], signedness-related range improvement [34], and memory operations precision improvement [57, 60, 65]. Due to the challenging nature of these analyses, these changes also led to bugs and security vulnerabilities [19, 21, 24, 29, 32, 38, 66], which sometimes are incorrectly fixed [67], or caused the feature to be dropped entirely [24].

[1]For simplicity, both helpers [3] and kfuncs [62] are referred to as helpers.

Agni [93] aims to automatically verify abstract operator soundness in BPF's range analysis. But verifying full soundness of the range analysis remains unsolved partly due to state pruning [16]. Additionally, applying Agni to the evolving BPF verifier also has scalability challenges due to its use of SMT solvers, which may increase the kernel maintenance burden to develop *efficiently* verifiable kernel code [15, 16], as the price for verified soundness.

Termination is another central safety property of BPF. BPF previously enforced a no-jump-backward property combined with a flat BPF program size limit. Over time, the need for looping prevailed, and several looping features were added [56, 73, 80, 85]. This further complicates safety reasoning because the verifier essentially needs to find an appropriate loop invariant to avoid having to reason about all possible looping iterations. However, generic automatic loop invariant reasoning is very complex [17, 41]. In the BPF verifier, loop invariant reasoning, partly represented as state-equivalence-based search pruning, is constantly updated for precision [25, 33, 74, 83], which naturally introduces bugs due to its complexity [20–23, 29, 35, 51, 84, 94, 97, 100–102].

3.2 Limited Programmability and Expressiveness

It is commonly reported that the verifier is unable to accept programs that are relatively simple and safe [39, 50], creating programmability and usability limitations for BPF users. Here, we discuss three notable manifestations of such limitations: the restriction for memory access patterns, the restriction on locking, and the restriction on loop complexity. These are a subset of the restrictions in BPF today, and more will inevitably appear as more features are added.

Restrictions on memory access patterns. The typical way for BPF programs to handle memory safety is through path-sensitive range analysis. The verifier make many approximations during the analysis, including: (1) discarding range information for content in BPF maps, (2) limiting numerical reasoning ability, and (3) forbidding dynamic access into stack or context objects. Under these constraints, developers are forced to perform additional checks and provide error handling logic at runtime, even if the program was safe without them, degrading performance and adding complexity [18, 37, 39, 45, 50, 78, 89]. Moreover, it is also commonly reported that the restrictions force users writing in high-level languages to use non-idiomatic code patterns that are difficult to maintain or migrate [1, 2].

Restrictions on locking. BPF programs are allowed to take spin locks to synchronize data accesses [86]. The verifier ensures that the program releases the lock on every execution path. However, the verifier implements a harsh solution to deadlocks — BPF programs can only a single lock at a time. Although doing so effectively eliminates the possibility of deadlocks [30], it also prevents developers from writing more complex synchronization patterns. Such restriction is difficult to work with in contexts such as BPF-extended scheduling, where BPF programs need to interact with multiple data structures concurrently [14]. The most promising systematic solution adds dynamic checks and changes to the locking semantics [39].

Restrictions on loop complexity. The verifier imposes limits on the number of instructions and branches it explores in an incoming program [87]. Programs exceeding these limits will be rejected. Such complexity limits exist not just to ensure program termination guarantee, but more importantly, to prevent the verifier from running indefinitely due to exponential path explosion.

As a result, developers of more complex BPF programs frequently find themselves fighting against the complexity limits. In many cases, developers have to refactor and split their programs into smaller pieces connected with BPF tail calls to pass verification, as shown by previous studies [50]. For example, the BPF Memcached Cache (BMC) [44] was split into seven programs, while logically two are required: networking ingress and egress.

The complexity limits also prevent BPF programs from having complex loops with non-trivial invariants [6, 39]. In BMC, the developers had to add an artificial packet length limit to ensure the loop iterating over the packet bytes would be accepted by the verifier [44].

4 Proposed Solution

4.1 KLean: System Design

We propose to use Lean as the safe kernel extension language. The overall architecture is shown in Figure 1. When the user wants to register a KLean extension on a KLean hook, they provides a Lean object representing an extension, whose type is specified by the hook. The type conformity (including safety properties) is checked by the Lean type checker. Then the checked Lean code will be invoked within the kernel whenever the hook point is triggered. KLean solves the limitations of static safety analysis by decomposing the problem into safety specification, safety checking, and safety proving. This decomposition simplifies the problem because it enables KLean to offload the most difficult and application-specific part — safety proving — entirely to user space.

Such offloading reduces the kernel's maintenance burden and lifts the restrictions on how users can write extension programs, provided they can prove them safe.

Safety specification. An important part of KLean is the interface specification between KLean and the kernel. Using Lean's powerful type system, KLean will annotate the KLean-kernel API (starting from BPF's helper functions) with type signatures that properly reflect the invariants for kernel safety, including those handled by the BPF verifier currently. We also envision the API to encode more complex

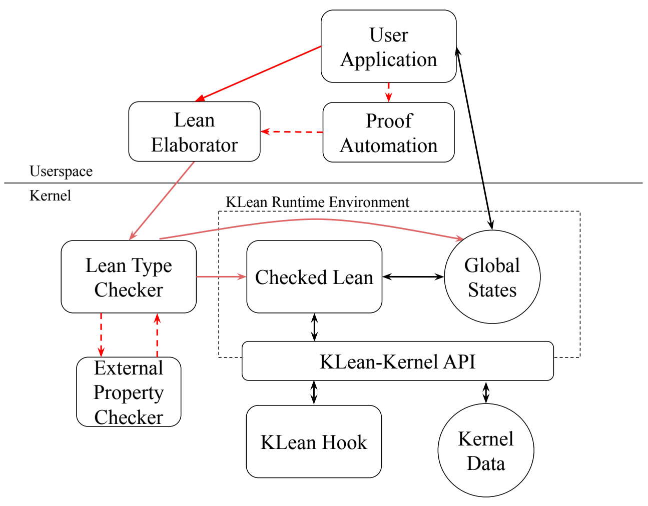

Figure 1. Overall architecture of KLean.

invariants (*e.g.,* locking order), allowing the kernel to expose lower-level APIs without wrapping them in additional runtime checks.

Safety checking. The powerful type system of Lean reduces safety checking to just type checking, which can be done by Lean's type checker. Relative to its generality, Lean type checker is very minimal. A Rust implementation of the type checker comprises only 7500 lines of code [7]. More importantly, based on well-studied mathematical theory, the Lean type system and type checker change rarely and minimally [4]. Therefore, KLean's verifier code will put minimal burden on OS kernel maintenance.

Safety proving. To write a safe KLean extension, the user bears the burden of creating an extension that Lean type checker will accept. Typically, this translates to writing an executable Lean function and providing a proof that it satisfies the safety properties encoded in the type requirement. As a result, the user is no longer restricted to certain code patterns, as long as they can provide a proof for the safety properties, unleashing expressiveness and freedom in extension development. Although safety proofs can require extra work from the users — which may be alleviated with proof automation (§5.1) — it does not restrict expressiveness because virtually all safe programs can be proven safe.[2]

4.2 Unlock Full Expressiveness

We now concretely discuss how KLean can handle the safety properties mentioned in Section 3.

[2]There is no guarantee that *every* safe program can be proven safe. But for programs that humans are able to reason to be safe, there is every reason to expect that the reasoning translates to Lean: for example, Lean's Mathlib library [68] has formalized a nontrivial portion of modern mathematics.

```
1  @[extern "k_read_array"]
2  opaque k_read_array (i : ℕ) (h: i < 16) : ℕ
3
4  theorem fermat_last_theorem (n a b c : ℕ)
5    (hn : 2 < n) (ha : 0 < a)
6    (hb : 0 < b) (hc : 0 < c)
7  : a ^ n + b ^ n ≠ c ^ n := by
8  -- Proved by Andrew Wiles and other mathematicians,
9  -- but the proof is too large to fit in the margin
10 done
11
12 structure PosInt where
13   val : ℕ
14   p : val > 0
15   -- other fields
16
17 def read_or_zero (n : ℕ) (a b c : PosInt) :
18   IO ℕ := do
19   if pred : a.val ^ n + b.val ^ n = c.val ^ n then
20     let x := k_read_array n (by
21       apply Nat.gt_of_not_le
22       intro n_gt_16
23       have n_gt_2 : n > 2 :=
24         Nat.lt_of_le_of_lt (by decide) n_gt_16
25       apply fermat_last_theorem n a.val b.val c.val
26             n_gt_2 a.p b.p c.p
27       exact pred
28       )
29     return x
30   else
31     return 0
```

Listing 1. Proving `k_read_array`'s range requirement using Fermat's last theorem.

Arbitrary Range Reasoning. Range requirement is commonly needed to ensure memory safety. In Listing 1 we show a hypothetical example where a kernel API `k_read_array` requires the index argument to be less than 16. In Lean, this property can be encoded as a proof argument `h` for proposition `i < 16`, along side the index argument `i`. Function `read_or_zero` takes one natural number n and three positive integers a, b, and c, only calling `k_read_array` with n if $a^n + b^n = c^n$. According to Fermat's last theorem, $n < 3$ is always true when such condition holds. Therefore the call is memory safe without needing additional checks. Such guarantee cannot be easily deduced by existing BPF infrastructure because (1) it is based on range property for composite data type `PosInt`, and (2) it has (extremely) nontrivial algebraic relations between variables that leads to the range limit on n.

KLean can admit this program because Lean's proof language is expressive enough to specify reasoning as complex as Fermat's last theorem (or other simpler safety proofs).

Importantly, all proofs (`k_read_array`'s `h` argument, `p` field in `PosInt`, and the entire proof between Line 20–28) are erased at runtime, meaning that the complex proof reasoning does not impact runtime performance at all. Apart from performance benefits, static reasoning is also preferred over dynamic checking because kernel error handling can sometimes be difficult (*e.g.,* holding a lock with inconsistent state changes).

Precise Locking Specification. BPF's current approach to deadlock-freedom is to only allow one lock held at any point. It is significantly more restrictive than the actual invariant the kernel maintains [71]: there is a strict global acquisition order between lock classes, and if a process wishes to hold multiple locks, they must be acquired in that order.

KLean can relax BPF's current restriction by precisely specifying such an invariant. To specify this property, KLean will first define a partial order between locks in a KLean extension. Then we can define the locking invariant by specifying properties over the locking operation history.

For example, in Listing 2, the KLean hook defines its desired KLean extension as type `LockSafeProg`, which includes two parts: `prog` that represents the actual program, and `p_order` that proves its lock acquisition order.

The executable `prog` will interact with the kernel states through the `KStateM` monad, an abstract class whose definition roughly corresponds to the available KLean-kernel interfaces (analogous to BPF helpers). `KStateM` can be instantiated with `RealKStateM` monad, where the interface is implemented by actual helpers, or `LockTraceStateM` monad, where the interface is implemented by stubs that trace the locking operations. The `p_order` is a proof that specifies if `prog` is plugged in with `LockTraceStateM`, the lock history it generates follows the lock acquisition order.

When the entire `LockSafeProg` is attached, the KLean hook extracts the executable function `prog` and instantiates it with `RealKStateM`, such that it will interact with the real environment. The entire proof reasoning, including the traced execution, proof over the locking traces, along with other proof components, will be erased at runtime, and no performance overhead is incurred.

```lean
section open KStateM
def myprog {m : Type → Type} [Monad m] [KStateM m]
  (n : ℕ) : m Unit := do
  lock 1
  let n ← get_v
  if n > 5 then
    lock 3
    set_v n+1
    unlock 3
  unlock 1
end

theorem myprog_lock_in_order : ∀ (s : KState),
  strictlyOrdered ((myprog TraceKStateM).run s).trace :=
  -- proof omitted

structure LockSafeProg where
  -- A program that interact with kernel state
  -- defined by the monad
  prog (m : Type → Type) [Monad m] [KStateM m] : m Unit
  -- For any input state, prog's lock operation
  -- satisfies the ordering condition
  p_order (s : KState):
    strictlyOrdered ((prog TracedKStateM).run s).trace
  -- omitting other safety properties

def hookTerm : LockSafeProg :=
  LockSafeProg.mk myprog myprog_lock_in_order
def hookRun : RealKStateM Unit := hookTerm.prog RealKStateM
```

Listing 2. Specifying a strict lock acquisition order over all possible execution traces in KLean.

Loop reasoning. Reasoning about loops or recursion does not add significant difficulty for KLean. The user can use Lean's type system to encode loop invariants or inductive hypotheses explicitly, without relying on implicit inference.

Termination is slightly more complicated. Well-typed Lean function always terminates, because Lean only allows well-founded recursions [58]. Therefore, the termination problem seems to be resolved for free. However, there is a crucial subtlety that makes this the more challenging property to specify: Although the OS kernel needs to avoid infinite loops, which is guaranteed by Lean automatically, it is also strongly desired for functions not to run too long in a preemption-disabled or interrupt-disabled context [63]. In fact, such problem exists in BPF as well: helper-based loops can cause significant stalling [51, 94].

Lean's type system, by design, cannot distinguish between two functions with the same input-output relations, meaning that execution time cannot be reasoned purely using Lean's type checker. We propose to use a hybrid approach of monad-based modeling [70] and an external property checker performing additional static analysis [47] external to Lean. Such design allows fine-grain user control of the reasoning while preventing malicious user extensions from escaping the resource accounting.

5 Research Directions

5.1 Improving KLean

We have shown that KLean provides major benefits over existing BPF infrastructure. However, BPF still has some benefits over KLean's design, and more research can be done to close the gap.

Improving trust-performance trade-off. Although BPF's verifier can be limiting for generic code patterns, for the most prominent BPF programs that the verifier targets, BPF can have better performance. This is because BPF programs are already optimized by the compiler [52, 64] before being verified and loaded into the kernel.

Lean is designed with performance in mind. However, adding optimizations post-verification would expand the TCB (trusted computing base). As shown in Figure 2, Lean currently has three evaluation modes with different trade-offs. Lean's most trusted execution is its core interpreter. However, it can be too slow for ordinary programs (*e.g.*, when they include non-structural recursions). The next option is native-extension-accelerated interpreter, where selected Lean types (*e.g.*, vectors, hash tables) have a native mirror implementation in C++. These extensions make the interpretation much more efficient, at the cost of trusting the correctness of their native implementation. Lastly, Lean supports compilation. It first runs a Lean-to-C compiler

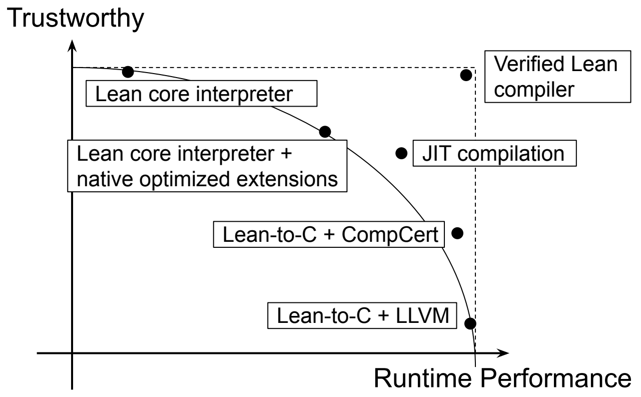

Figure 2. Lean execution model and trade-offs.

written in Lean, then compiles the C code using a bundled LLVM compiler. Compilation comes at the cost of trusting the entire Lean-to-C compiler and LLVM compiler under potentially malicious input, which is beyond the capability of typical kernel maintaining groups.

Ideally, we would create a verified Lean compiler that maintains all the safety guarantees while producing fast native code. Before that is possible, we propose that the first version of KLean should adopt Lean's accelerated interpreter execution model. We also propose two research directions to improve the trust-performance trade-off: (1) Modify the Lean-to-C compiler to target the C dialect supported by verified C compilers like CompCert [61], and (2) a baseline just-in-time compilation using boxed Lean objects with type- and hint-guided inlining and specialization.

DSL and proof automation. One benefit of BPF, despite confusion and problems [50], is its automated safety reasoning. If developers follow a restricted code pattern, safety can be proven without additional effort.

In the general scenario, finding safety proofs automatically is undecidable. However, it is possible to bring BPF's philosophy into KLean. For scenarios where a restricted language suffices, such as regular expression or circuit-like language [28, 79], Lean's powerful meta-programming capability can define such a language as a DSL within Lean and automatically derive safety proofs. Because the DSLs still translate to Lean, the kernel-side infrastructure does not need any changes to support new DSLs.

It is also possible to create DSLs like BPF with best-effort proof automation, providing a similar interface to existing BPF development (à la Nelson et al. [76]). Then, the original verification strategy of the BPF verifier can be added as proof tactics, allowing the user to fill in the gaps when the BPF verifier algorithm fails. Importantly, the verification engine for BPF can now reside in user space, allowing it to be freely extended without changing the safety assurances provided by KLean. Instead, if a bug were introduced to the BPF verifier tactic, KLean would simply reject the generated proof, eliminating the risk to kernel safety.

5.2 Advancing OS Kernel Extension Research

KLean will enable a wide range of operating-system-related research projects that can be difficult to efficiently implement in BPF due to its various restrictions. We will discuss some preliminary ideas that demonstrate such potential.

Efficient system call virtualization. A class of systems, including MBOX [54], CDE [46], and others [9, 36, 42], modifies system call semantics to enhance security, visibility, performance, and compatibility in off-the-shelf software. These systems transform, record, or block system calls to achieve their objectives. However, they either require kernel changes or introduce unnecessary overhead using process-level tracing [8], which induce additional context switching and data-copying because of the extra round trip on each system call.

With KLean, it will be possible to implement system call virtualization: customize system call semantics by building KLean extensions on top of the current kernel. Research systems, in the same class as MBOX and CDE, could be implemented in this manner, getting the best of both worlds.

Complete data-path delegation. Many applications [44, 98] with service-based architecture use BPF to speed up their fast-path processing, avoiding excessive context switching and data-copying. However, due to BPF's verifier restrictions, much of the complex request handling logic still has to remain in user space. With KLean's much more powerful safety and termination verification capabilities, we envision that it is feasible to delegate entire data paths to the kernel without sacrificing any safety, while only keeping the control-path management in the user space process.

KLean-aware compiler optimization. Past research on BPF-enhanced networking [98, 99] and storage [96] has shown the power of kernel extension on data-intensive tasks. Leveraging KLean's expressiveness, we propose to investigate a opportunistic compiler optimization: transforming applications snippets into kernel extensions, automatically improving their performance without sacrificing safety.

For instance, applied to distributed or middlebox systems, the compiler can try to identify network-heavy code snippets that do not expose intermediate results with the rest of the system. Then the optimization will amalgamate the system calls (similar to Anycall [43]) and transform them into KLean extensions. Such opportunistic optimization, guided by heuristics and dynamic profiling, can improve network performance automatically while allowing developers to program with more focus on the logical abstraction.

6 Related Work

Requiring proofs for the safety of kernel extensions is not new. In fact, safe kernel extensions for packet filtering are one of the main proposed use cases for proof-carrying code [75]. Building on decades of advances in formal verification in theory and software systems, KLean offers more powerful and well-defined specification and proving capabilites.

Verus [59] is a verification framework that enables specifying and proving properties about Rust programs, utilizing a SMT solver for its proof checking. KLean chooses Lean because proof terms are first class construct in the language, which enables clear and simple kernel-user interface design.

Rex [50] proposes an alternative safe kernel extension scheme based on Rust. It simplifies safety verification by leveraging Rust's type system and safety guarantees. Unlike KLean, however, Rex cannot easily specify more custom safety properties without extending its core. For example, Rex cannot enforce complex invariants such as locking order and termination without runtime components.

Formally verified operating systems such as seL4 [55] aim to improve kernel reliability with formal verification. KLean complements this effort by enabling run-time extensions to be formally verified. We also envision KLean to enable more kernel development in the form of safe kernel extensions, achieving partial verification in giant monolithic kernels like Linux.

7 Conclusion

We propose KLean, a Lean framework for implementing safe kernel extensions, leveraging the power of a dependent type proof assistant. Compared to BPF, the existing approach, KLean enables the decoupling and offloading of safety proving responsibility to userspace, reducing kernel maintenence burden and improving extension expressiveness.

Acknowledgments

We are thankful to anonymous PLOS reviewers for their input. This material is based upon research supported by NSF awards CNS-2247687, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] Access packet data as &[u8] from XdpContext. https://github.com/aya-rs/aya/issues/100. (Nov. 2021).
[2] BPF Verifier errors when using aya logger with strings, c_chars, and [u8]. https://github.com/aya-rs/aya/issues/546. (Mar. 2023).
[3] `bpf-helpers(7)` – Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.
[4] Lean git history. https://github.com/leanprover/lean4/commits/master/src/kernel/type_checker.cpp.
[5] `libbpf`. https://github.com/libbpf/libbpf.
[6] missing btf func_info whilst using bpf_loop(). https://github.com/aya-rs/aya/issues/521. (Feb. 2023).
[7] nanoda_lib. https://github.com/ammkrn/nanoda_lib/.
[8] `ptrace(2)` – Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html.
[9] Windows Subsystem for Linux. https://github.com/microsoft/WSL.
[10] CVE-2017-17864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17864, December 2017.
[11] CVE-2018-18445. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18445, October 2018.
[12] CVE-2020-8835. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835, February 2020.
[13] CVE-2021-3490. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490, April 2021.
[14] Alden, D. A new type of spinlock for the BPF subsystem. https://lwn.net/Articles/1016674/.
[15] Alden, D. Formally verifying the BPF verifier. https://lwn.net/Articles/1020664/.
[16] Alden, D. Verifying the BPF verifier's path-exploration logic. https://lwn.net/Articles/1021825/.
[17] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. 51*, 3 (2018).
[18] Borkmann, D. bpf: compress maglev per service lut into flat slot id array. https://github.com/cilium/cilium/commit/bfaef16f34851aada5189df2eaebce351dd2543e. (Sept. 2020).
[19] Borkmann, D. bpf: Fix alu32 const subreg bound tracking on bitwise operations. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=049c4e13714ecbca567b4d5f6d563f05d431c80e. (May. 2021).
[20] Borkmann, D. bpf: Fix check_return_code to only allow [0,1] in trace_iter progs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2ec0616e870f0f2aa8353e0de057f0c2dc8d52d5. (May. 2020).
[21] Borkmann, D. bpf: Fix incorrect verifier pruning due to missing register precision taints. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=71b547f561247897a0a14f3082730156c0533fed. (Apr. 2023).
[22] Borkmann, D. bpf: Fix pointer arithmetic mask tightening under state pruning. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e042aa532c84d18ff13291d00620502ce7a38dda. (Jul. 2021).
[23] Borkmann, D. bpf: Fix verifier jmp32 pruning decision logic. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fd675184fc7abfd1e1c52d23e8e900676b5a1c1a. (Feb. 2021).
[24] Borkmann, D. bpf: Undo incorrect __reg_bound_offset32 handling. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f2d67fec0b43edce8c416101cdc52e71145b5fef. (Mar. 2020).
[25] Borkmann, D. bpf, verifier: further improve search pruning. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=07016151a446d25397b24588df4ed5cf777a69bb. (Aug. 2016).
[26] Bove, A., Dybjer, P., and Norell, U. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics* (2009), Springer, pp. 73–78.
[27] Cao, X., Patel, S., Lim, S. Y., Han, X., and Pasquier, T. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (2024), pp. 369–378.
[28] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: a

declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1987), pp. 178–188.

[29] Chaignon, P. bpf: Fix incorrect state pruning for <8B spill/fill. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=345e004d023343d38088fdfea39688aa11e06ccf. (Dec. 2021).

[30] Coffman, E. G., Elphick, M., and Shoshani, A. System deadlocks. *ACM Computing Surveys (CSUR) 3*, 2 (1971), 67–78.

[31] Corbet, J. A JIT for packet filters. https://lwn.net/Articles/437981/.

[32] Cree, E. bpf/verifier: fix bounds calculation on BPF_RSH. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4374f256ce8182019353c0c639bb8d0695b4c941. (Dec. 2017).

[33] Cree, E. bpf/verifier: track liveness for pruning. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=dc503a8ad98474ea0073a1c5c4d9f18cb8dd0dbf. (Aug. 2017).

[34] Cree, E. bpf/verifier: track signed and unsigned min/max values. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b03c9f9fdc37dab81ea04d5dacdc5995d4c224c2. (Aug. 2017).

[35] Cree, E. bpf/verifier: when pruning a branch, ignore its write marks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=63f45f840634ab5fd71bbc07acff915277764068. (Aug. 2017).

[36] Curtsinger, C., and Barowy, D. W. Riker:Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 885–898.

[37] Decina, A. Minor tweaks to make the verifier's job easier. https://github.com/aya-rs/aya/commit/2ac433449cdea32f10c8fc882187999995946032d. (Jul. 2022).

[38] Dwivedi, K. K. bpf: Fix state pruning for STACK_DYNPTR stack slots. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d6fefa1105dacc8a742cdcf2f4bfb501c9e61349. (Jan. 2023).

[39] Dwivedi, K. K., Iyer, R., and Kashyap, S. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP'24)* (Nov. 2024).

[40] Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017).

[41] Furia, C. A., Meyer, B., and Velder, S. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR) 46*, 3 (2014), 1–51.

[42] Gaidis, A. J., Atlidakis, V., and Kemerlis, V. P. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM Conference on Computer and Communications Security (CCS)* (2023).

[43] Gerhorst, L., Herzog, B., Reif, S., Schröder-Preikschat, W., and Hönig, T. Anycall: Fast and flexible system-call aggregation. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (2021), pp. 1–8.

[44] 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).

[45] Graf, T. bpf: Fix verifier error when writing to skb->cb[0]. https://github.com/cilium/cilium/commit/1e25adb69b44573ecaea42803fff7312fbfe9372. (May. 2023).

[46] Guo, P. J., and Engler, D. CDE: Using system call interposition to automatically create portable software packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)* (2011).

[47] Hoffmann, J., and Shao, Z. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032* (2015), Springer-Verlag, p. 132–157.

[48] Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P., and Kozyrakis, C. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *ACM Symposium on Operating Systems Principles (SOSP)* (2021), pp. 588–604.

[49] IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[50] Jia, J., Qin, R., Craun, M., Lukiyanov, E., Bansal, A., Phan, M., Le, M. V., Franke, H., Jamjoom, H., Xu, T., and Williams, D. Rex: Closing the Language-Verifier Gap with Safe and Usable Kernel Extensions. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC'25)* (July 2025).

[51] Jia, J., Sahu, R., Oswald, A., Williams, D., Le, M. V., and Xu, T. Kernel extension verification is untenable. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2023), pp. 150–157.

[52] Jose E. Marchesi. eBPF support for GCC. https://gcc.gnu.org/legacy-ml/gcc-patches/2019-08/msg01987.html.

[53] Kaffes, K., Humphries, J. T., Mazières, D., and Kozyrakis, C. Syrup: User-Defined Scheduling Across the Stack. In *ACM Symposium on Operating Systems Principles (SOSP)* (2021), pp. 605–620.

[54] Kim, T., and Zeldovich, N. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 139–144.

[55] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an os kernel. SOSP '09, Association for Computing Machinery, p. 207–220.

[56] Koong, J. bpf: Add bpf_loop helper. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e6f2dd0f80674e9d5960337b3e9c2a242441b326.

[57] Koong, J. bpf: Add verifier support for dynptrs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=97e03f521050c092919591e668107b3d69c5f426. (May. 2022).

[58] Kunen, K. *Set Theory an Introduction to Independence Proofs*, vol. 102. Elsevier, 2014.

[59] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages 7*, OOPSLA1 (2023), 286–315.

[60] Lau, M. K. bpf: Support <8-byte scalar spill and refill. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=354e8f1970f821d4952458f77b1ab6c3eb24d530. (Sep. 2021).

[61] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress* (2016).

[62] Linux Developers. BPF Kernel Functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html.

[63] Linux Kernel. Softlockup detector and hardlockup detector (aka nmi_watchdog). https://docs.kernel.org/admin-guide/lockup-watchdogs.html.

[64] LLVM Project. LLVM 3.7 Release Notes. https://releases.llvm.org/3.7.0/docs/ReleaseNotes.html#non-comprehensive-list-of-changes-in-this-release.

[65] Matei, A. bpf: Allow variable-offset stack access. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=01f810ace9ed37255f27608a0864abebccf0aab3. (Dec. 2023).

[66] Matei, A. bpf: Fix verification of indirect var-off stack access. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a833a17aeac73b33f79433d7cee68d5cafd71e4f. (Dec. 2023).

[67] Matei, A. bpf: Protect against int overflow for stack access size. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ecc6a2101840177e57c925c102d2d29f260d37c8. (Mar. 2024).

[68] mathlib Community, T. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2020), CPP 2020, Association for Computing Machinery, p. 367–381.

[69] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference* (1993).

[70] McCarthy, J., Fetscher, B., New, M. S., Feltey, D., and Findler, R. B. A coq library for internal verification of running-times. *Science of Computer Programming 164* (2018), 49–65.

[71] Molnar, I., and van de Ven, A. Runtime locking correctness validator. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[72] Moura, L. d., and Ullrich, S. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28* (Cham, 2021), A. Platzer and G. Sutcliffe, Eds., Springer International Publishing, pp. 625–635.

[73] Nakryiko, A. bpf: add support for open-coded iterator loops. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=06accc8779c1d558a5b5a21f2ac82b0c95827ddd.

[74] Nakryiko, A. bpf: decouple prune and jump points. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bffdeaa8a5af7200b0e74c9d5a41167f86626a36. (Dec. 2022).

[75] Necula, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1997), pp. 106–119.

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

[77] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[78] Reimerink, D. bpf,test: Fix verifier issues in IPv6 BPF tests when running locally. https://github.com/cilium/cilium/commit/ceaa4c42b0101c69d82407046e8437942237edcb. (May. 2023).

[79] snarky developers. Ocaml dsl for verifiable computation. https://github.com/o1-labs/snarky.

[80] Song, Y. bpf: Add bpf_for_each_map_elem() helper. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=69c087ba6225b574afb6e505b72cb75242a3d844.

[81] Song, Y. bpf: Fix a verifier failure with xor. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2921c90d471889242c24cff529043afb378937fa. (Aug. 2020).

[82] Song, Y. bpf: Provide better register bounds after jmp32 instructions. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=581738a681b6faae5725c2555439189ca81c0f1f. (Nov. 2019).

[83] Starovoitov, A. bpf: add search pruning optimization to verifier. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f1bca824dabba4ffe8582f87ca587780befce7ad. (Sept. 2014).

[84] Starovoitov, A. bpf: fix callees pruning callers. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eea1c227b9e9bad295e8ef984004a9acf12bb68c. (Jun. 2019).

[85] Starovoitov, A. bpf: introduce bounded loops. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2589726d12a1b12eaaa93c7f1ea64287e383c7a5.

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

[87] Starovoitov, A. bpf: verifier (add docs). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=51580e798cb61b0fc63fa3aa6c5c975375aa0550. (Sept. 2014).

[88] Starovoitov, A. precise scalar_value tracking. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b5dc0163d8fd78e64a7e21f309cf932fda34353e. (Jun. 2019).

[89] Sun, H., and Su, Z. Lazy Abstraction Refinement with Proof. In *Linux Plumbers Conference (LPC'24)*. https://lpc.events/event/18/contributions/1939/. (Sept. 2024).

[90] The Cilium Authors. Cilium: eBPF-based Networking, Observability, Security. https://cilium.io/.

[91] The Rocq Development Team. *The Rocq Prover Reference Manual*. Inria, 2025. Version 9.0.0.

[92] Ullrich, S., and de Moura, L. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang. 6*, ICFP (Aug. 2022).

[93] Vishwanathan, H., Shachnai, M., Narayana, S., and Nagarakatte, S. Verifying the verifier: ebpf range analysis verification. In *Computer Aided Verification* (2023), C. Enea and A. Lal, Eds., Springer Nature Switzerland, pp. 226–251.

[94] Williams, D., and Sahu, R. When bpf programs need to die: exploring the design space for early bpf termination. In *Linux Plumbers Conference* (2023).

[95] Yelam, A., Wu, K., Guo, Z., Yang, S., Shashidhara, R., Xu, W., Novaković, S., Snoeren, A. C., and Keeton, K. PageFlex: Flexible and efficient user-space delegation of linux paging policies with eBPF. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (2025), pp. 291–306.

[96] Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., et al. XRP: In-Kernel Storage Functions with eBPF. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2022), pp. 375–393.

[97] Zhou, C. bpf: Relax allowlist for css_task iter. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3091b667498b0a212e760e1033e5f9b8c33a948f. (Oct. 2023).

[98] Zhou, Y., Wang, Z., Dharanipragada, S., and Yu, M. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 1391–1407.

[99] Zhou, Y., Xiang, X., Kiley, M., Dharanipragada, S., and Yu, M. DINT: Fast In-Kernel distributed transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 401–417.

[100] Zingerman, E. bpf: correct loop detection for iterators convergence. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2a0992829ea3864939d917a5c7b48be6629c6217. (Oct. 2023).

[101] Zingerman, E. bpf: exact states comparison for iterator convergence checks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2793a8b015f7f1caadb9bce9c63dc659f7522676. (Oct. 2023).

[102] Zingerman, E. bpf: Fix for use-after-free bug in inline_bpf_loop.

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fb4e3b33e3e7f13befdf9ee232e34818c6cc5fb9. (Jun. 2022).