KLean: Extending Operating System Kernels with Lean

Di Jin[1] Ethan Lavi[1] Jinghao Jia[2] Robert Y. Lewis[1] Nikos Vasilakis[1]

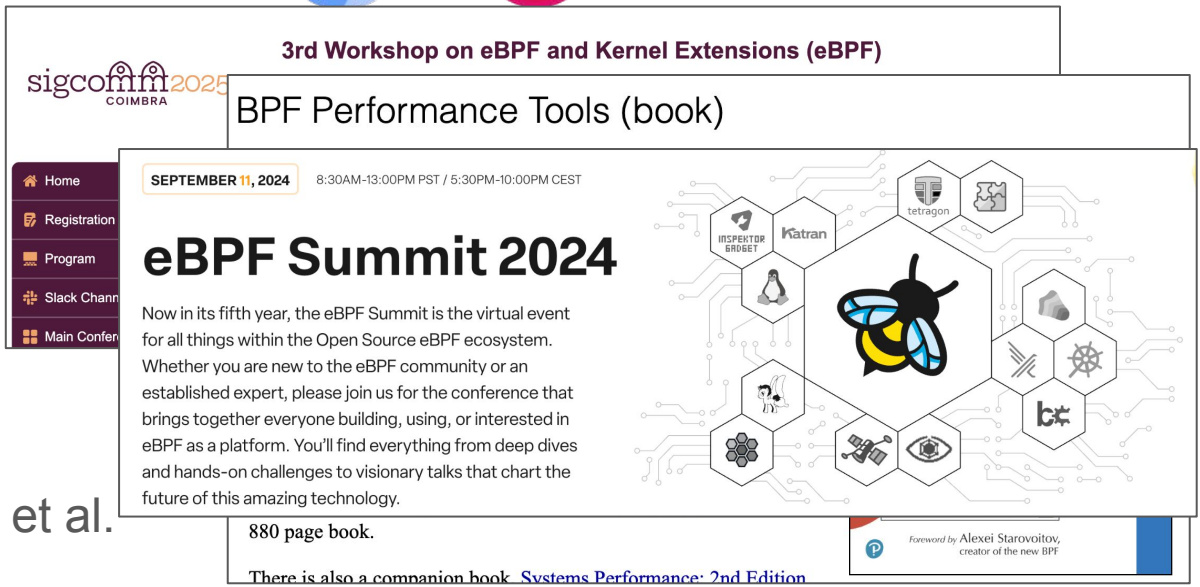
[1]Brown University
[2]University of Illinois Urbana-Champaign

BPF: Linux's Safe Extension Language

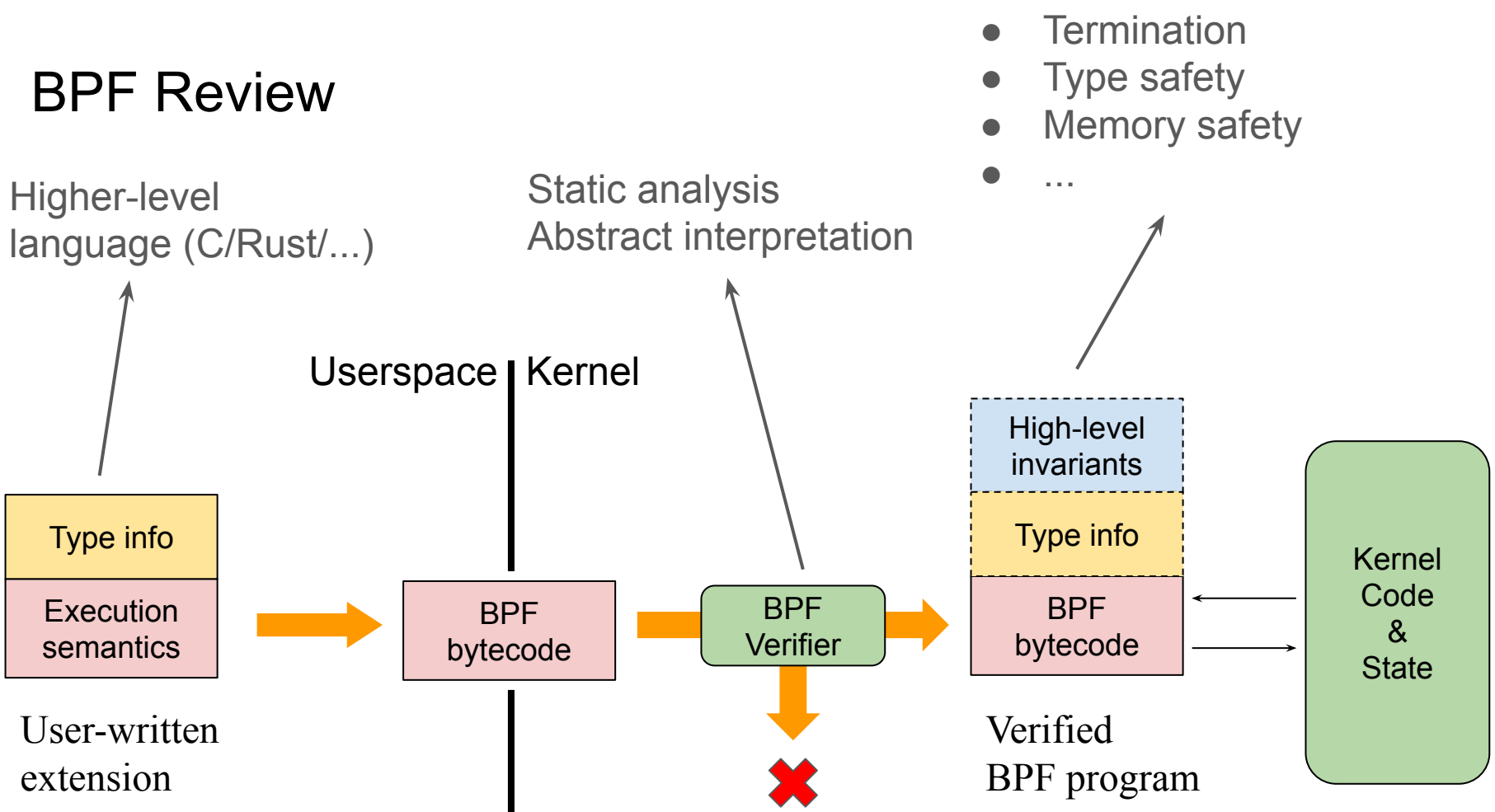
Safe customizability without the cost of **context switching** or **data movement**

- Networking
- Profiling
- Security
- Storage
- Scheduling

BMC [NSDI'21] by Ghigoff et al.
XRP [OSDI'22] by Zhong et al.
ghOSt [SOSP'21] by Humphries et al.
...



BPF Review


Higher-level language (C/Rust/...)
Static analysis
Abstract interpretation
Termination
Type safety
Memory safety
...
Userspace
Kernel
Type info
Execution semantics
BPF bytecode
BPF Verifier
High-level invariants
Type info
BPF bytecode
Kernel Code & State
User-written extension
Verified BPF program

BPF's Vision

"eBPF is a crazy technology – like putting JavaScript into the Linux kernel..." --- Brendan Gregg

"BPF programs are safe and portable kernel modules" --- Alexei Starovoitov

"Think of eBPF as a new type of software which bridges the gap between a typical monolithic kernel and microkernel..." --- Daniel Borkmann

Battling the Verifier

"I want to build a low-latency KV store.
Let me use BPF for this."

Battling the Verifier

Unintuitive error with obscure error message

"Wait...why is my perfect code rejected?"

```
48: (2d) if r3 > r2 goto pc+8          ;
R2_w=pkt_end(off=0,imm=0) R3_w=pkt(off=34,r=34,imm=0)
; return (void *)(unsigned long)ctx->data_end;
49: (67) r2 <<= 32
R2 pointer arithmetic on pkt_end prohibited
processed 45 insns (limit 1000000) max_states_per_insn 1
total_states 4 peak_states 4 mark_read 2
-- END PROG LOAD LOG --
libbpf: prog 'handle_ipv4_from_netdev': failed to load: -13
libbpf: failed to load object
'/home/user/projects/cilium/cilium/bpf/tests/abpf.o'
Error: failed to load object file
```

```
if (index < (pkt_end - ptr)/4)
```

❌

```
if (ptr + 4*index < pkt_end)
```

✅

Battling the Verifier

"But my program has just 400 instructions."

```
; for (__u16 j = 0; j < MAX_SERVER_NAME_LENGTH; j++) {
76: (25) if r3 > 0xfb goto pc+3
77: (07) r3 += 1
78: (07) r4 += 8
79: (3d) if r1 >= r4 goto pc-15
65: (bf) r4 = r2
66: (0f) r4 += r3
67: (71) r5 = *(u8 *)(r4 +6)
BPF program is too large. Processed 1000001 insn
processed 1000001 insns (limit 1000000) max_states_per_insn
34 total_states 10376 peak_states 7503 mark_read 3
```

Analysis can **time out** for small programs with **complex control flow**

Battling the Verifier

"Just need to submit a kernel patch for better analysis."

BPF verifier precision tracking improvements

From: Andrii Nakryiko <andrii-AT-kernel.org>
To: <bpf-AT-vger.kernel.org>, <ast-AT-kernel.org>, <daniel-AT-iogearbox.net>
Subje
Date:
Mess
Cc:
Archi

This p
SCALAR

Patche

Patch
This v
subpro

Patch

author Martin KaFai Lau <kafai@fb.com> 2021-09-21 17:49:41 -0700
committer Alexei Starovoitov <ast@kernel.org> 2021-09-26 13:07:27 -0700
commit 354e8f1970f821d4952458f77b1ab6c3eb24d530 (patch)
tree aa7b59f3430c464970949823bb235eaddaf092f4
parent 27113c59b6d0a587b29ae72d4ff3f832f58b0651 (diff)
download linux-354e8f1970f821d4952458f77b1ab6c3eb24d530.tar.gz

bpf: Support <8-byte scalar spill and refill

The verifier currently does not save the reg state when

Subject: [PATCH] bpf, verifier: Improve precision for BPF_ADD and BPF_SUB
Date: Tue, 10 Jun 2025 18:13:55 -0400 [thread overview]
Message-ID: <20250610221356.2663491-1-harishankar.vishwanathan@gmail.com> (raw)

This patch improves the precison of the scalar(32)_min_max_add and
scalar(32)_min_max_sub functions, which update the u(32)min/u(32)_max
ranges for the BPF_ADD and BPF_SUB instructions. We discovered this more
precise operator using a technique we are developing for automatically
synthesizing functions for updating tnums and ranges.

According to the BPF ISA [1], "Underflow and overflow are allowed during
arithmetic operations, meaning the 64-bit or 32-bit value will wrap".
Our patch leverages the wrap-around semantics of unsigned overflow and
underflow to improve precision.

Below is an example of our patch for scalar_min_max_add; the idea is
analogous for all four functions.

Battling the Verifier

"Just need to fix this kernel vulnerability I created"

CVE-2023-2163 Detail

MODIFIED

CVE-2022-23222 Detail

MODIFIED

CVE-2017-17864 Detail

Description

kernel/bpf/verifier.c in the Linux kernel through 4.14.8 mishandles states_equal comparisons between the pointer data type and the UNKNOWN_VALUE data type, which allows local users to obtain potentially sensitive address information, aka a "pointer leak."

Metrics

CVSS Version 4.0 | CVSS Version 3.x | CVSS Version 2.0

NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.

CVSS 3.x Severity and Vector Strings:

NIST: NVD **Base Score:** 3.3 LOW **Vector:** CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N

References to Advisories, Solutions, and Tools

By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have

Battling the Verifier

Usability problem: BPF programs are difficult to develop

- Code patterns can get rejected due to verifier's inaccuracy
  - Pointer arithmetics restricted for the accuracy of points-to analysis
  - Control-flow complexity restricted by analysis time limit
  - **...**

Maintenance burden problem: the kernel is difficult to maintain

- Constant refinements for analysis accuracy
- Bugs caused by the frequent changes

Rethink the Design of Safe Kernel Extensions

BPF's design demands a **sound** static analysis to be **accurate**, which is fundamentally hard.

Soundness: accepted extensions are safe

Accuracy: safe extensions are accepted (maximally)

Decouple the obligations using a **proof-carrying language**

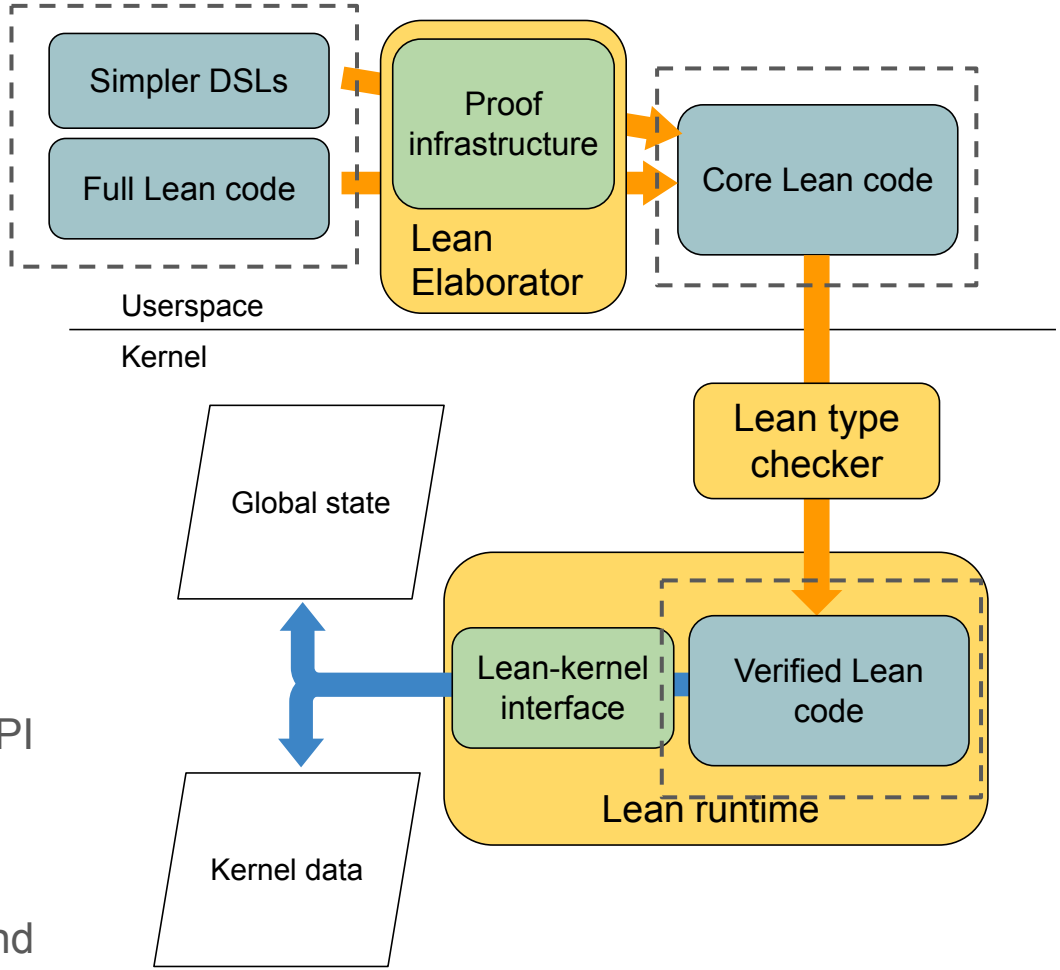
Soundness ⇔ **safety specification** & **checking** → Kernel: maintenance burden ⤵

Accuracy ⇔ **safety proving** → Userspace: usability ⤴

KLean: Proposed Design

We propose KLean: a safe kernel extension framework using Lean

- In-kernel Lean environment
  - Type checker
  - Runtime
- Lean-kernel interface
  - Lean specification of kernel API exposed for extension
- Proof infrastructure
  - Domain-specific languages and proof tactics



KLean: Proposed Design

KLean.lean

```
1  class KStateM (m : Type → Type) [Monad m] where
2    get_random_n : m UInt32
3    printk (s : String) : m Unit
4    lock (l : Lock) : m Unit
5    -- more defs
6
7  class XDPExt where
8    prog (m : Type → Type) (p : Pkt) [KStateM m] : m Action
9    prog_lock_correct : lockOrdered prog
```

MyExt.lean

```
1  import KLean
2  def myext (m : Type → Type) (p : Pkt) [KStateM m] : m Action := do
3    let x ← KStateM.get_random_n
4    KStateM.printk s!"Hello {x}"
5    return Action.Pass
6  theorem myprog_lock_correct := ...
7  def extMain : XDPExt := {
8    prog := myprog
9    prog_lock_correct := myprog_lock_correct
10 }
```

Kernel-state-accessing API

Expected function signature

Expected proof for overall safety properties

Implementing executable function

Providing proof

Register KLean extension

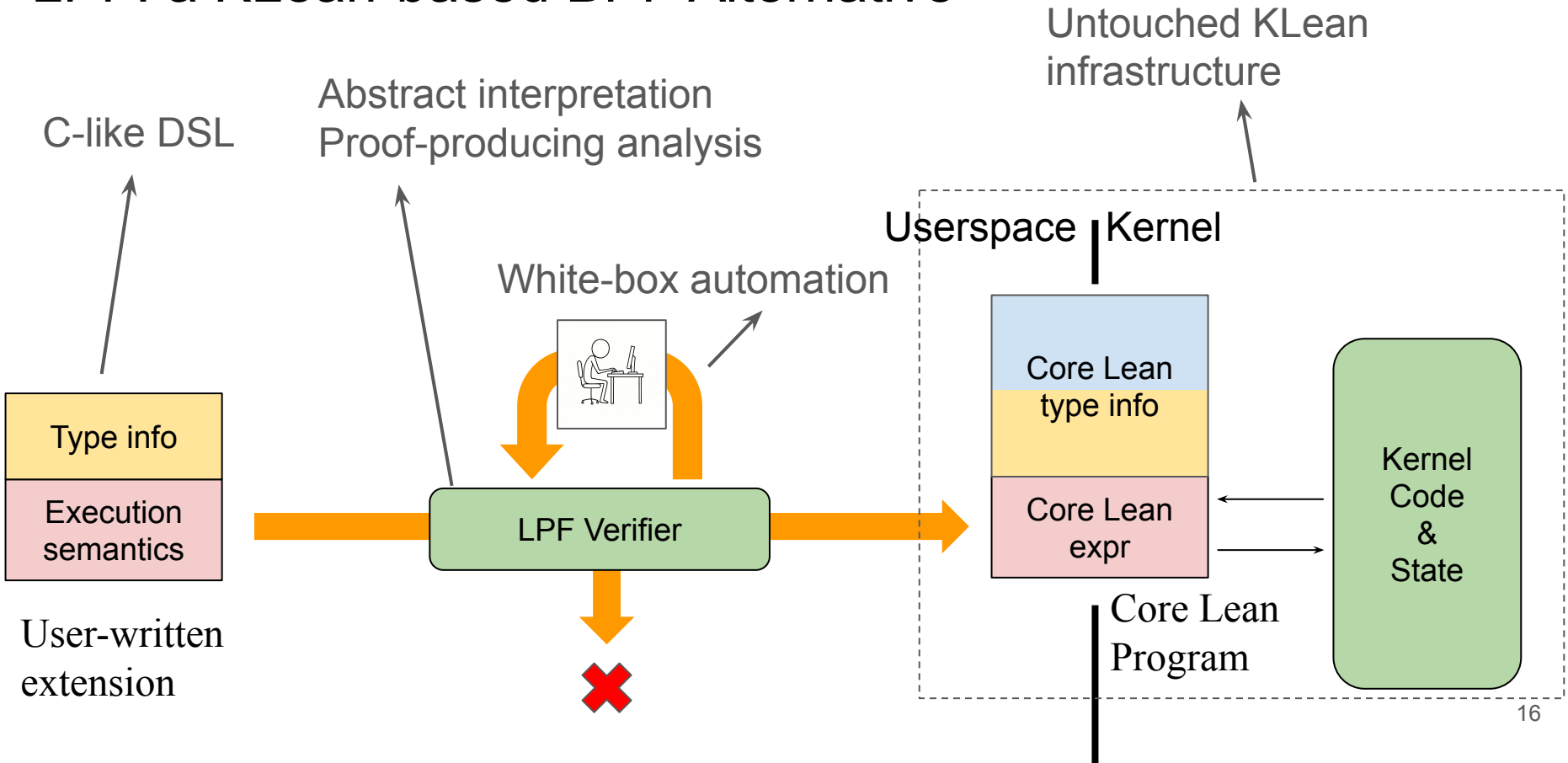
KLean's Kernel-side Benefits

- **Less maintenance burden**
  - Lean type checker as verifier --- relatively minimal
  - Decidable and (mostly) efficient algorithm
  - Rare changes (except Lean-kernel interface)
- Better specification language --- CIC type system
  - **Decouples** specification from verification
  - Allows writing **modular** and **reusable** specifications
  - Allows specifying **complex safety properties** (e.g. locking order)

KLean's Userspace-side Benefits

- **Better usability**: more expressive and well-defined language
  - \+ No unintuitive restrictions on code patterns
  - \+ No control-flow complexity limit
  - − No automatic in-kernel safety proving
    - ↳ This is a plus (with some work) for KLean:
      we can still do automation in userspace
- Proof infrastructure --- implemented through meta-programming
  - Domain-specific languages and proof tactics
  - BPF-style full proof automation

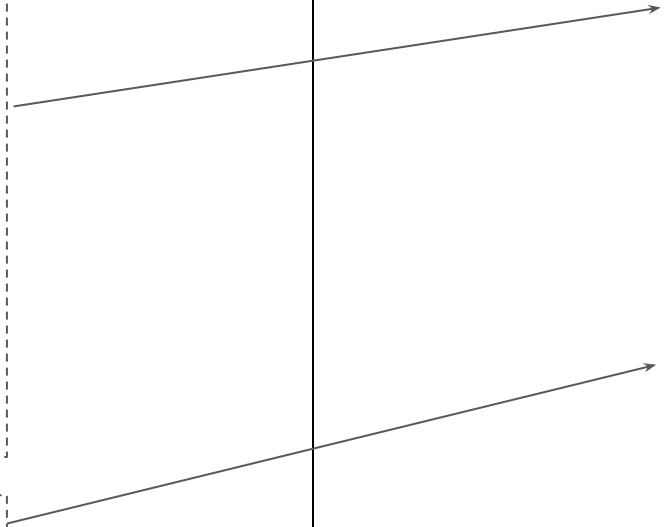
LPF: a KLean-based BPF Alternative



LPF: a KLean-based BPF Alternative

MyExt.lean

```
1  let c_module := C_PROG_START
2  void swap(int *a, int *b) {
3      *a = *a + *b;
4      *b = *a - *b;
5      *a = *a - *b;
6  }
7  int main(void *ctx) {
8      ...
9      swap(&x, &y);
10     z = ip[x];
11     ...
12 }
13 C_PROG_DONE
14
15 let m := lpf_verify c_module
16 ...
```



C-like DSL

Lowering and proof generation

LPF: a KLean-based BPF Alternative

MyExt.lean

```
1  let c_module := C_PROG_START
2  void swap(int *a, int *b) {
3      *a = *a + *b;
4      *b = *a - *b;
5      *a = *a - *b;
6  }
7  int main(void *ctx) {
8      ...
9      swap(&x, &y);
10     z = ip[x];
11     ...
12 }
13 C_PROG_DONE
14
15 let m := lpf_verify c_module
16 ...
```

*a: [0, 3]	*b: [0, 3]	
*a: [0, 6]	*b: [0, 3]	
*a: [0, 6]	*b: [-3, 6]	
*a: [-3, 9]	*b: [-3, 6]	

x: [0, 3]	y: [0, 3]	ip: size 4
x: [-6, 9]	y: [-3, 6]	ip: size 4

Invalid access

LPF: a KLean-based BPF Alternative

MyExt.lean

```
1  let c_module := C_PROG_START
2  void swap(int *a, int *b) {
3      *a = *a + *b;
4      *b = *a - *b;
5      *a = *a - *b;
6  }
7  int main(void *ctx) {
8      ...
9      swap(&x, &y);
10     z = ip[x];
11     ...
12 }
13 C_PROG_DONE
14
15 theorem swap_range : ...
16 let m := lpf_verify c_module [swap_range]
17 ...
```

*a: [0, 3]	*b: [0, 3]	
*a: [0, 6]	*b: [0, 3]	
*a: [0, 6]	*b: [-3, 6]	
*a: [-3, 9]	*b: [-3, 6]	

x: [0, 3]	y: [0, 3]	ip: size 4
x: [-6, 9]	y: [-3, 6]	ip: size 4

Invalid access

LPF: a KLean-based BPF Alternative

MyExt.lean

```
1  let c_module := C_PROG_START
2  void swap(int *a, int *b) {
3      *a = *a + *b;
4      *b = *a - *b;
5      *a = *a - *b;
6  }
7  int main(void *ctx) {
8      ...
9      swap(&x, &y);
10     z = ip[x];
11     ...
12 }
13 C_PROG_DONE
14
15 theorem swap_range : ...
16 let m := lpf_verify c_module [swap_range]
17 ...
```

*a: [0, 3] *b: [0, 3]

*a: [0, 6] *b: [0, 3]

*a: [0, 6] *b: [-3, 6]

*a: [-3, 9] *b: [-3, 6]

x: [0, 3] y: [0, 3] ip: size 4

x: [0, 3] y: [0, 3] ip: size 4

x: [0, 3] y: [0, 3] z: [0, 255] ip: size 4

Things We Skipped

Additional benefits

- Lean's Imperative-friendly IR and other optimization tricks
- Built-in verified data structure (e.g. hash table)

Other challenges

- Bounding time and space complexity
- Performance-TCB tradeoff

Potential applications

- Efficient system call virtualization
- Trustworthy user-implemented drivers
- Complete data path delegation

Summary

We propose KLean: a safe OS kernel extension framework using Lean, which includes

- **In-kernel Lean environment** for safety checking and program execution
- **Lean-kernel interface** for safety specification
- **Proof infrastructure** in user space for easier/automatic safety proving

and such design will

- improve **usability** by providing intuitive programming interface and helpful proof tactics
- reduce kernel **maintenance burden** by simplifying safety specification as well as safety verification

Find Us

Di Jin

Ethan Lavi

Jinghao Jia

Robert Y. Lewis

Nikos Vasilakis

Backup slides

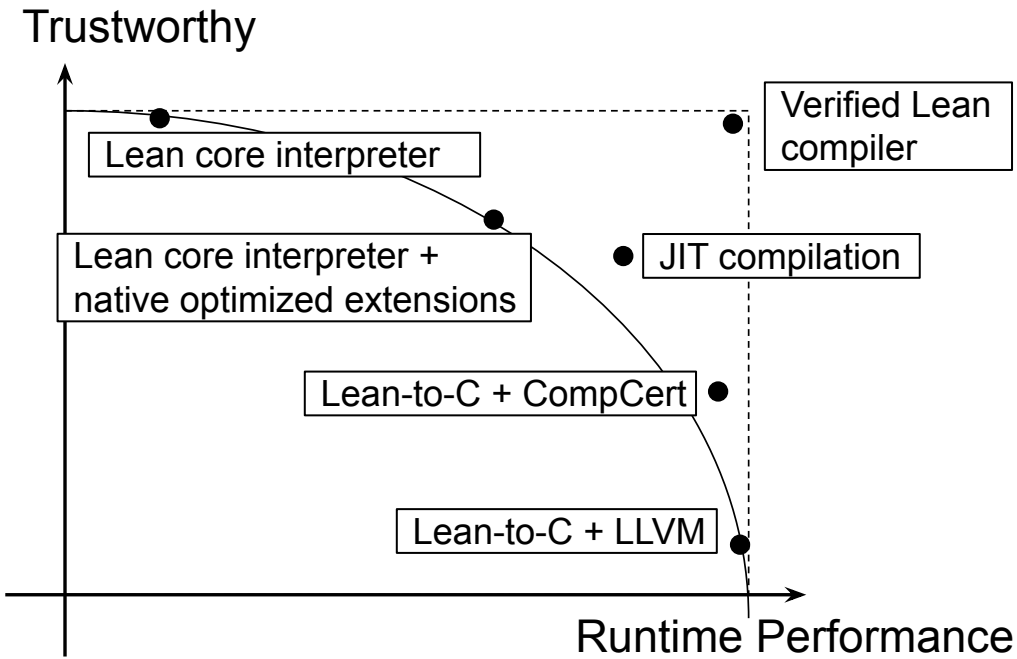
Challenges: Performance

Enemies

- Frequent memory allocation
- Costly abstraction mechanism
- TCB size restriction

Allies

- Lean's imperative-friendly IR and optimization
- Implementation shadowing



Simplifying Kernel-side Verification

Subsystem	LoC	Bug fix/total commits	Ratio
mm	130k	2404/24522	9.8%
sched	35k	532/4671	11.4%
net/core	64k	1311/10051	13%
fs/ext4	47k	390/5354	7.3%
bpf	**57k**	**910/3945**	**23.1%**

Linux kernel bug fix commits by subsystem (at version 6.16-rc6)

Core Lean type checker

- Decidable and efficient algorithm
- Multiple external implementations
  - *e.g.,* 7.8k Rust LoC
- Much less maintenance burden

  --- Lean's core type system rarely changes