

HiveGuard: Proof-Carrying eBPF Verification

Martin Fink, Dennis Sprokholt, Reto Achermann, Pramod Bhatotia

Systems Research Group

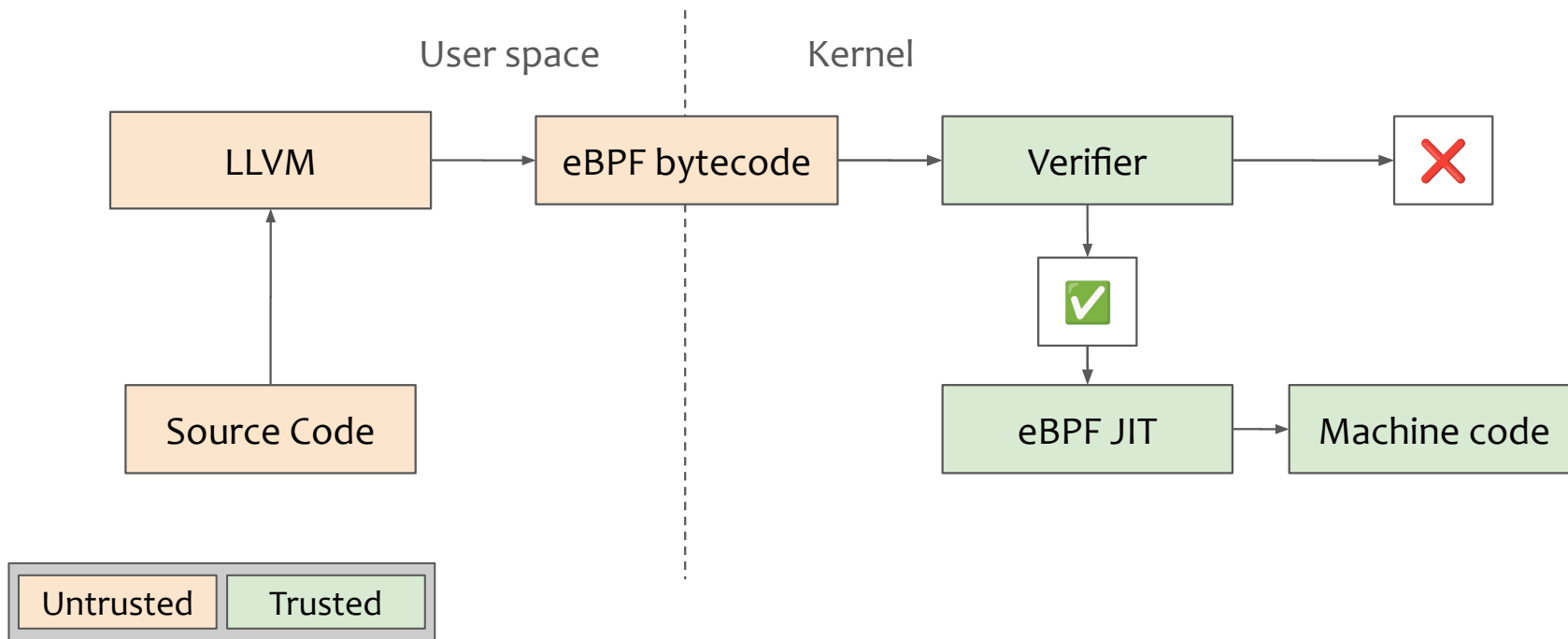
<https://dse.in.tum.de/>

Linux Plumbers Conference 2026

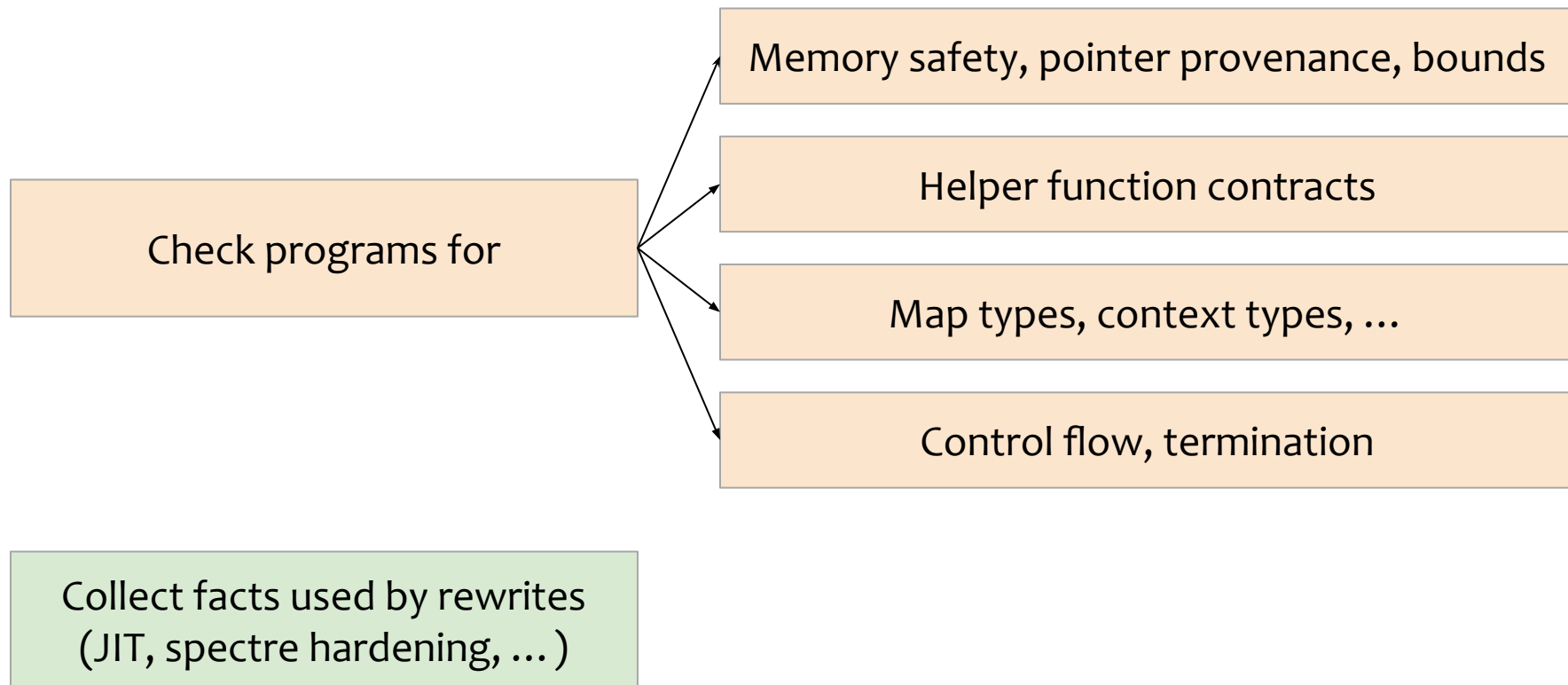


The Linux eBPF Verifier

- Threat model: userspace is considered untrusted
- Verifier has to discover safety arguments at load-time



What the Verifier is Responsible For

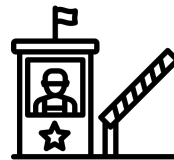


Drawbacks of In-Kernel-Verification



Large trusted analysis

The in-kernel verifier is
~21.3k LoC (Sep 2026)



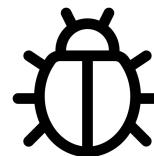
Verifier is a security boundary

Soundness bugs exist [1,2] and threaten
the kernel's integrity



Imprecision has usability consequences

Safe programs are rejected, requiring
safety workarounds



eBPF JIT compiler is trusted

Code transformation after verification is
trusted and not checked

[1] Jia et al. "Kernel Extension Verification Is Untenable." HotOS'23.

[2] Okelmann et al. "MorphOS: An Extensible Networked Operating System." CoNEXT'25.

What can we do?

Improve the analysis [1, 2, 3]

- + Better safety reasoning
- Large in-kernel verifier

Runtime defenses [4, 5, 6, 7]

- + Unverifiable programs can run
- + Protect against verifier bugs
- Runtime cost

[1] Gershuni et al. "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions." PLDI'19.

[2] Hamza et al. "A Flow-Sensitive Refinement Type System for Verifying eBPF Programs." OOPSLA'25.

[3] Sun et al. "Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement." SOSP'25.

[4] Dwivedi et al. "Fast, Flexible, and Practical Kernel Extensions." SOSP'24.

[5] Jia et al. "Kernel Extension Verification Is Untenable." HotOS'23.

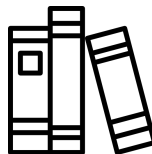
[6] Okelmann et al. "MorphOS: An Extensible Networked Operating System." CoNEXT'25.

[7] Lim et al. "SafeBPF: Hardware-Assisted Defense-in-Depth for eBPF Kernel Extensions." CCSW'24.

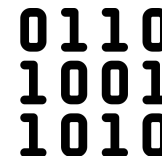
How can we build a verifier architecture that decouples complex safety reasoning from the trusted mechanism that enforces its results?



Efficient verification



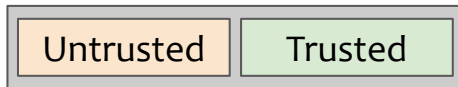
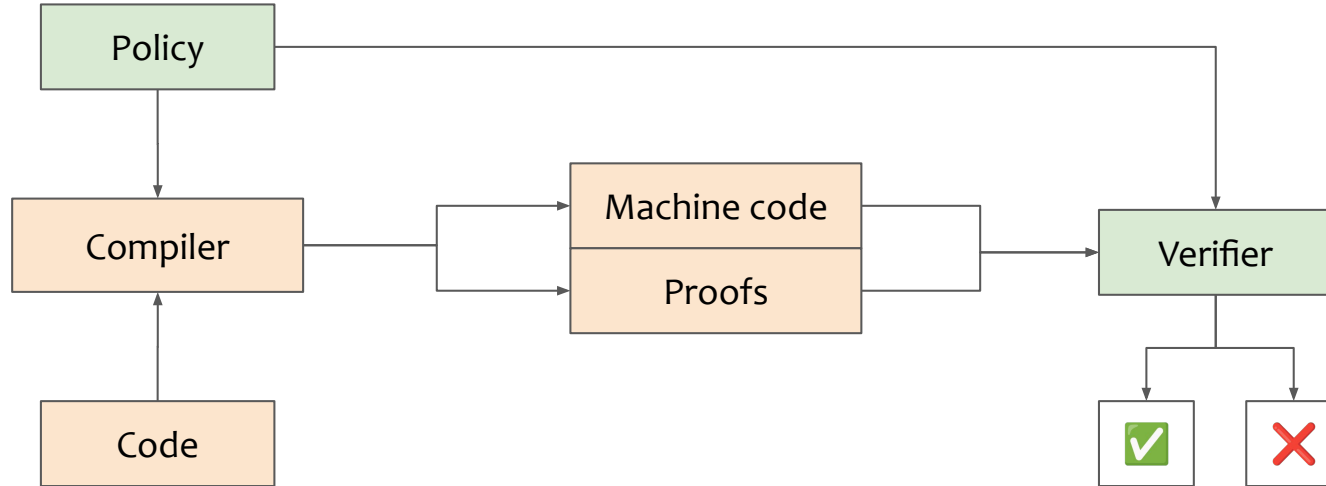
Real-world programs



Post-JIT verification

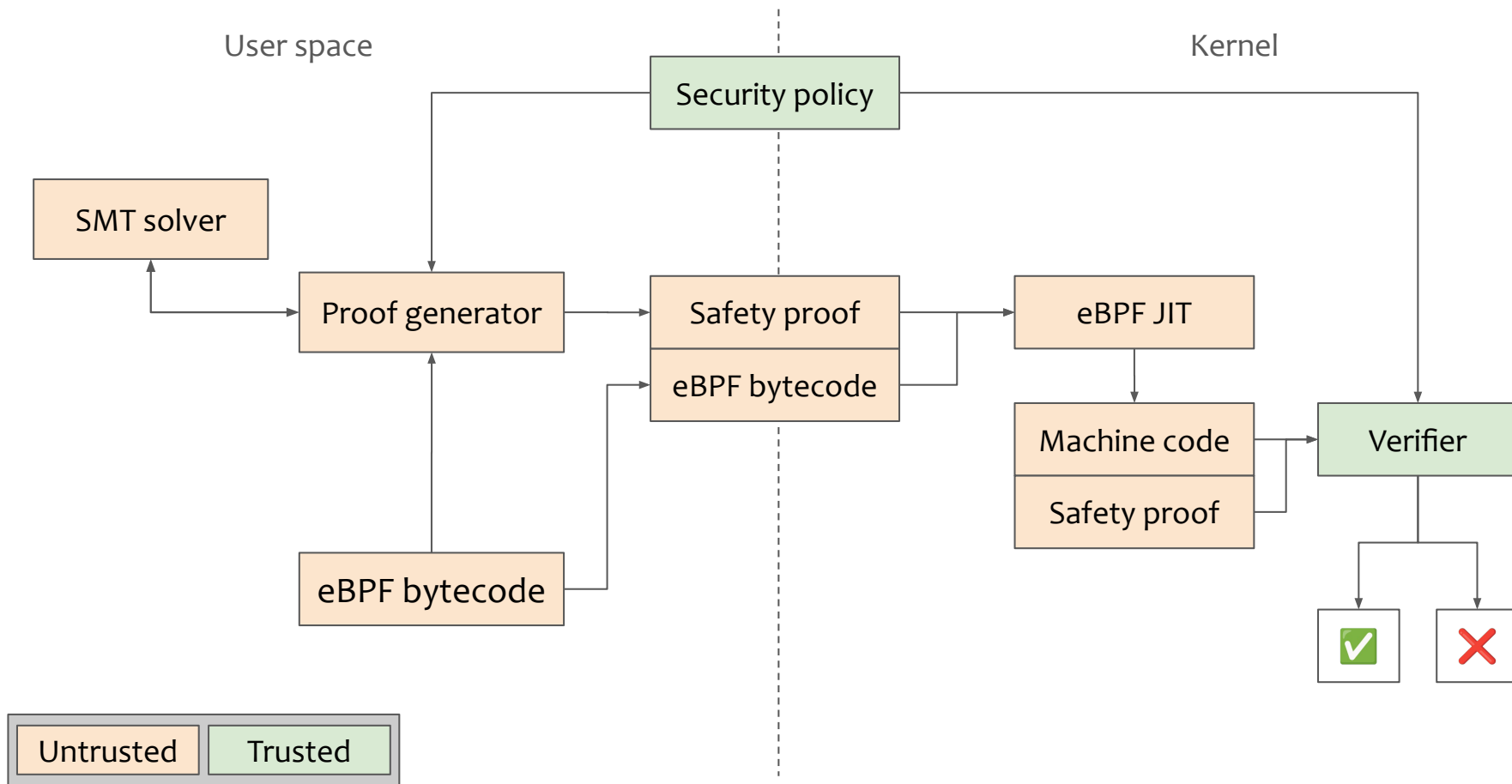
Apply **Proof-Carrying Code** [8] and produce proofs in userspace, verifying them in the kernel!

Background: Proof-Carrying Code (PCC) [8]

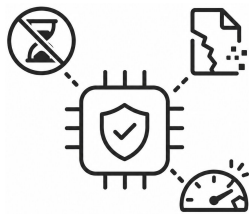


[8]: Necula, George C. "Proof-Carrying Code." POPL '97

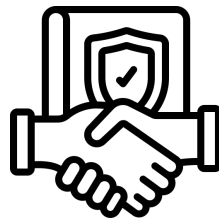
HiveGuard: Proof-Carrying eBPF Verification



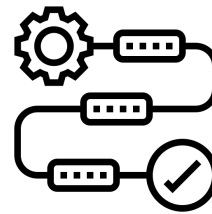
Challenges Applying PCC to eBPF



In-kernel verification



Proof generator and
verifier must agree



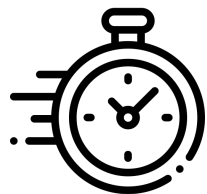
JIT must transform
proofs

Userspace is **untrusted** and **unconstrained**



Userspace is **untrusted**:

It may craft malicious proofs and try to exhaust the verifiers resources



Userspace is **unconstrained**:

It may run expensive analyses, e.g. using SMT solvers

Proof language must be
expressive

Proof checking must be *fast*

We propose a proof language that can reason about

- Pointers
- Ranges, Congruences
- Comparisons
- Memory contents

Ranges

$r1 \in [0, 63]$

Congruences

$r2 \bmod 8 == 0$

Comparisons

$r3 <_u r4$ $r3 >_s r4$

Pointers

```
r1 = ptr(ctx, nonnull, off=0, size=8192, align=1, perm=r)
r10 = ptr(stack(frame0), nonnull, off=0, size=512, align=8, perm=rw)
```

Offset

Size of memory region

read/write permissions

Type of pointer (ctx, stack, maps, ...)

Nullability

Alignment of pointer

Memory initialization

`mem stack(frame0) [-8, 0) = unknown`

stack is initialized

Memory facts

`mem map_value(map0) [0, 8)  $\in$  [0, 63]`

map entry contains known value

How Can a Constrained Proof Language Prove Enough?



- Assertions are attached at block entries
- They describe facts needed to prove a block correct, but do not describe **how new facts** are derived

```
unsigned i = ...;
```

```
invariant:
```

```
0 <= i <= 0xFFFF_FFFF
```

```
i &= 0xFF;
```

```
if (i < 255) {
```

```
    int[256] mem;  
    mem[i] = 0;
```

```
}
```

How do we know where
`i < 256` comes from?

Constrained Proof Language: Hints

Untrusted hints select from a trusted lemma catalogue

```
hint edge pc 7 -> pc 8: NarrowNonNull(r0 != 0, r0 = ptr(...))
```

The name of the lemma

Where to apply the hint

The lemma's inputs

The checker:

1. Verifies the lemma's inputs
2. Reads the instruction at the site
3. Computes the conclusion

```
NarrowNonNull: p != 0, p = ptr(...) ⊢ p = ptr(nonnull, ...)
```

Other examples:

1. Deriving a bitmask bound ($r0 \ \&= \ 7$)
MaskBoundConst: $r0 \in [0, 7]$
2. Narrowing from comparison (`if r2 < 64 goto <pc>`)
NarrowUpperFromLt: $r2 \text{ <u } 64 \vdash r0 \in [0, 63]$

Hints are untrusted: Their conclusion comes from the trusted verifier, not from the proof!

Constrained Proof Language: Example

```
SEC("socket")
int foo(struct __sk_buff *skb)
{
    __u32 key = 0;
    __u64 *p = bpf_map_lookup_elem(&map, &key);
    if (!p)
        return 0;

    __u32 n = skb->len;
    if (n > 8)
        return 0;

    __u64 sum = 0;
    for (__u32 i = 0; i < n; i++)
        sum += p[i];
    return sum;
}
```

Constrained Proof Language: Example



```
__u32 key = 0;
__u64 *p = bpf_map_lookup_elem(&map, &key);
if (!p)
    return 0;

__u32 n = skb->len;
if (n > 8)
    return 0;

__u64 sum = 0;
for (__u32 i = 0; i < n; i++)
    sum += p[i];
return sum;
```

```
<foo>:
; invariant at pc 0:
;   r1 = ptr(ctx, nonnull, perm=r, ...)
;   r10 = ptr(stack(frame0), nonnull, size=512, perm=rw, ...)
...
1: w6 = 0x0
...

7: call 0x1 // bpf_map_lookup_elem(map, &key)
; r0 = ptr(map_value(map0), maybe_null, ...)

8: if r0 == 0x0 goto <pc 23>
; hint edge pc 8 -> pc 9: NarrowNonNull(r0 != 0, r0 = ptr(...))

; invariant at pc 9:
;   r0 = ptr(map_value(map0), nonnull, ...)
...
22: w0 = w6
23: exit
```

Constrained Proof Language: Example



```
__u32 key = 0;
__u64 *p = bpf_map_lookup_elem(&map, &key);
if (!p)
    return 0;

__u32 n = skb->len;
if (n > 8)
    return 0;

__u64 sum = 0;
for (__u32 i = 0; i < n; i++)
    sum += p[i];
return sum;
```

```
<foo>:
; invariant at pc 0:
;   r1  = ptr(ctx, nonnull, perm=r, ...)
;   r10 = ptr(stack(frame0), nonnull, size=512, perm=rw, ...)
...
1:  w6 = 0x0
...

7:  call 0x1    // bpf_map_lookup_elem(map, &key)
; r0 = ptr(map_value(map0), maybe_null, ...)

8:  if r0 == 0x0 goto <pc 23>
; hint edge pc 8 -> pc 9: NarrowNonNull(r0 != 0, r0 = ptr(...))

; invariant at pc 9:
;   r0 = ptr(map_value(map0), nonnull, ...)
...
23: w0 = w6
23: exit
```

Constrained Proof Language: Example



```
__u32 key  
__u64 *p =  
if (!p)
```

"Is this branch/operation
relevant to the verifier?"

```
    return 0;
```

```
__u32 n = skb->len;  
if (n > 8)  
    return 0;
```

```
__u64 sum = 0;  
for (__u32 i = 0; i < n; i++)  
    sum += p[i];
```

```
<foo>:  
; invariant at pc 0:  
;   r1 = ptr(ctx, nonnull, perm=r, ...)  
;   r10 = ptr(stack(frame0), nonnull, size=512, perm=rw, ...)  
...  
1:  w6 = 0x0  
...  
  
7:  call 0x1    // bpf_map_lookup_elem(map, &key)  
;   r0 = ptr(map_value(map0), maybe_null, ...)  
  
8:  if r0 == 0x0 goto <pc 23>  
;   hint edge pc 8 -> pc 9: NarrowNonNull(r0 != 0, r0 = ptr(...))  
  
; invariant at pc 9:  
;   r0 = ptr(map_value(map0), nonnull, ...)  
...  
22: w0 = w6  
23: exit
```

NarrowNonNull: $p \neq 0, p = \text{ptr}(\dots) \vdash p = \text{ptr}(\text{nonnull}, \dots)$

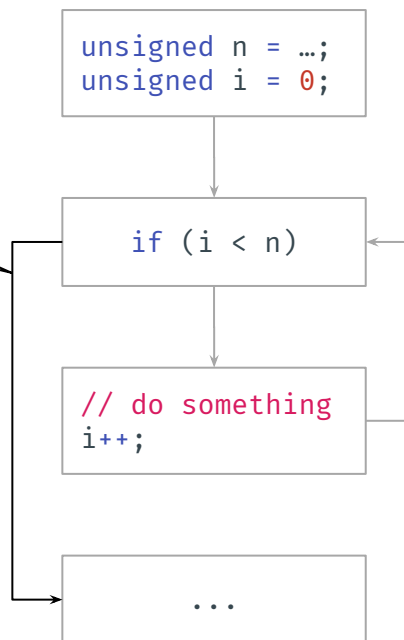
Loop Termination

How can we prove loop termination without exhaustive state exploration?

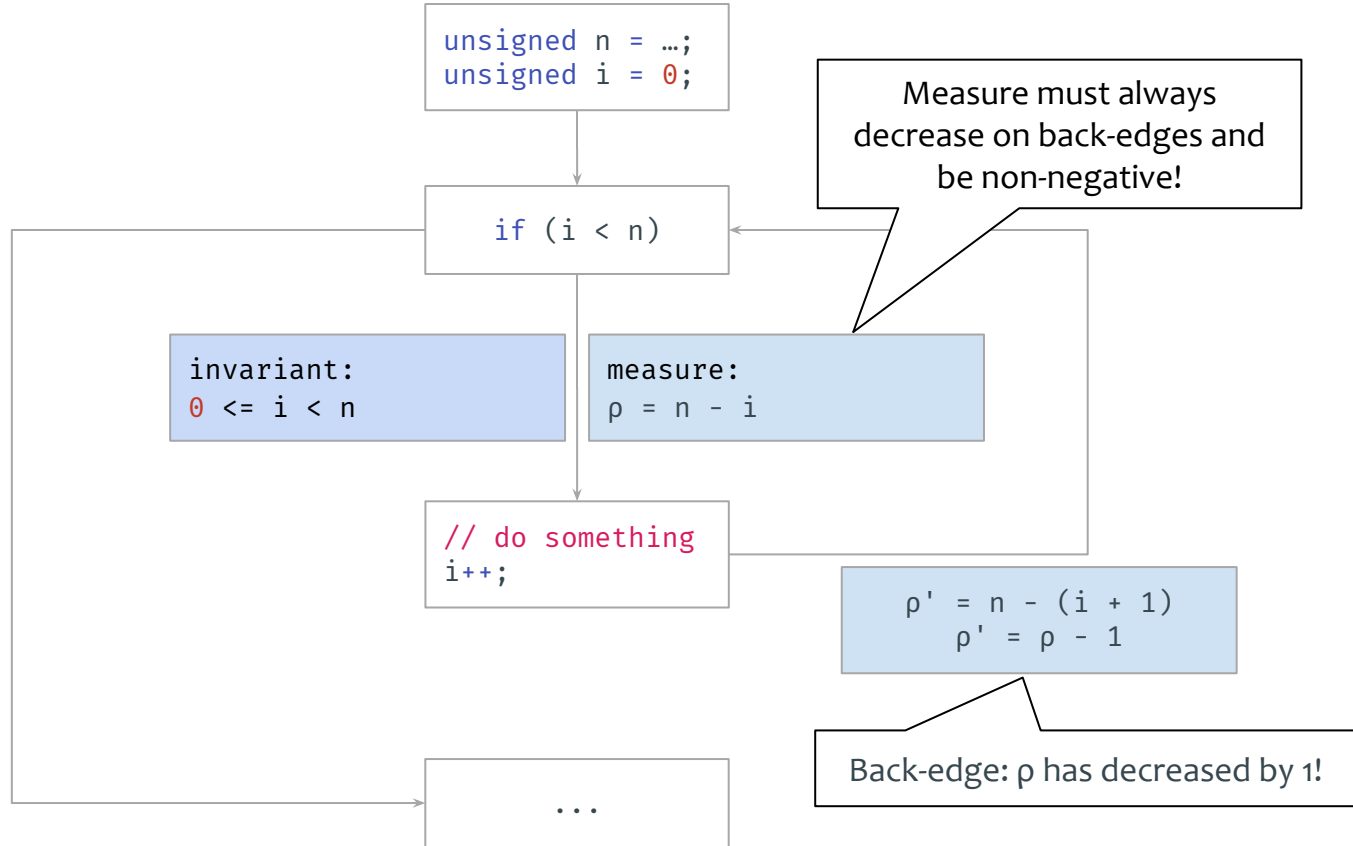
```
unsigned n = ...;  
for (unsigned i = 0; i < n; i++) {  
    // do something  
}
```

How do we know this branch is eventually taken?

Generate a linear ranking function [9] for every back edge!



Loop Termination



Loop Termination: Concrete Example

```
SEC("socket")
int foo(struct __sk_buff *skb)
{
    __u32 key = 0;
    __u64 *p = bpf_map_lookup_elem(&map, &key);
    if (!p)
        return 0;

    __u32 n = skb->len;
    if (n > 8)
        return 0;

    __u64 sum = 0;
    for (__u32 i = 0; i < n; i++)
        sum += p[i];
    return sum;
}
```

Loop Termination: Concrete Example

```
__u32 key = 0;
__u64 *p = bpf_map_lookup_elem(&map, &key);
if (!p)
    return 0;

__u32 n = skb->len;
if (n > 8)
    return 0;

__u64 sum = 0;
for (__u32 i = 0; i < n; i++)
    sum += p[i];
return sum;
```

```
<foo>:
...

; invariant at pc 14:
;   r1 ∈ [1, 8] // n
;   r2 ∈ [0, 7] // i
;   r0 = ptr(...)
; measure:
;   7 - r2
14:  r3 = *(u64 *)(r0 + 0x0) // load p[i]
15:  r6 += r3                // sum += r3
16:  r0 += 0x8               // move pointer to p[i + 1]
17:  r2 += 0x1               // measure decreases by one
18:  if r2 < r1 goto <pc 14>
; hint edge pc 18 -> pc 14: NarrowUpperFromLt(r2 < r1)

19:  w0 = w6
20:  exit
```

Invariant when entering the
loop body

Loop Termination: Concrete Example

```
__u32 key = 0;
__u64 *p = bpf_map_lookup_elem(&map, &key);
if (!p)
    return 0;

__u32 n = skb->len;
if (n > 8)
    return 0;

__u64 sum = 0;
for (__u32 i = 0; i < n; i++)
    sum += p[i];
return sum;
```

```
<foo>:
...
```

```
; invariant at pc 14:
;   r1 ∈ [1, 8] // n
;   r2 ∈ [0, 7] // i
;   r0 = ptr(...)
; measure:
;   7 - r2
```

```
14: r3 = *(u64 *)(r0 + 0x0) // load p[i]
15: r6 += r3                // sum += r3
16: r0 += 0x8               // move pointer to p[i + 1]
17: r2 += 0x1               // measure decreases by one
18: if r2 < r1 goto <pc 14>
; hint edge pc 18 -> pc 14: NarrowUpperFromLt(r2 < r1)

19: w0 = w6
20: exit
```

Measure: $0x7 - i$

Loop Termination: Concrete Example

```
__u32 key = 0;
__u64 *p = bpf_map_lookup_elem(&map, &key);
if (!p)
    return 0;

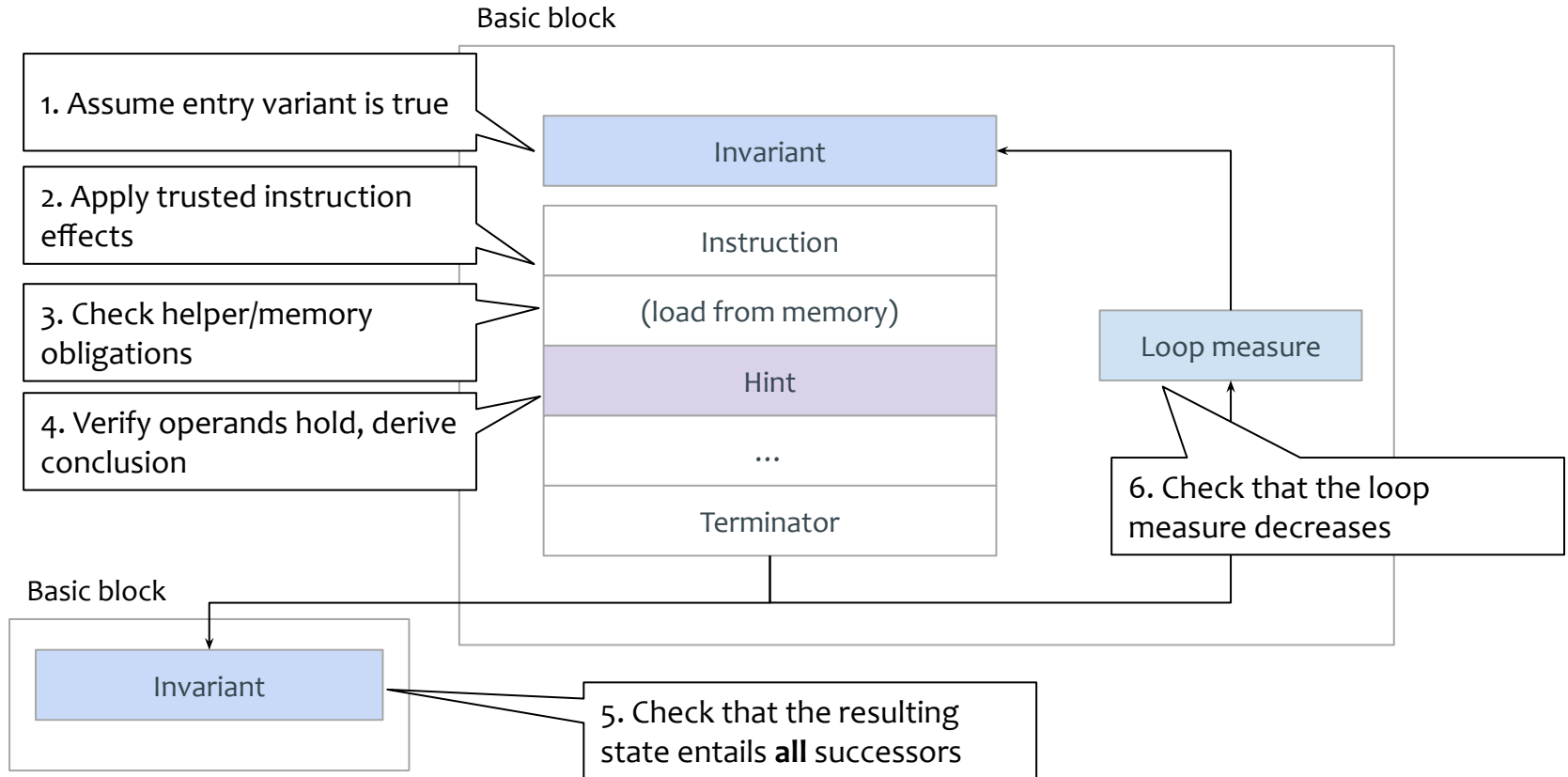
__u32 n = skb->len;
if (n > 8)
    return 0;

__u64 sum = 0;
for (__u32 i = 0; i < n; i++)
    sum += p[i];
return sum;
```

```
<foo>:
...
; invariant at pc 14:
;   r1 ∈ [1, 8] // n
;   r2 ∈ [0, 7] // i
;   r0 = ptr(...)
; measure:
;   7 - r2
14: r3 = *(u64 *)(r0 + 0x0) // load p[i]
15: r6 += r3                // sum += r3
16: r0 += 0x8               // move pointer to p[i + 1]
17: r2 += 0x1               // measure decreases by one
18: if r2 < r1 goto <pc 14>
; hint edge pc 18 -> pc 14: NarrowUpperFromLt(r2 < r1)

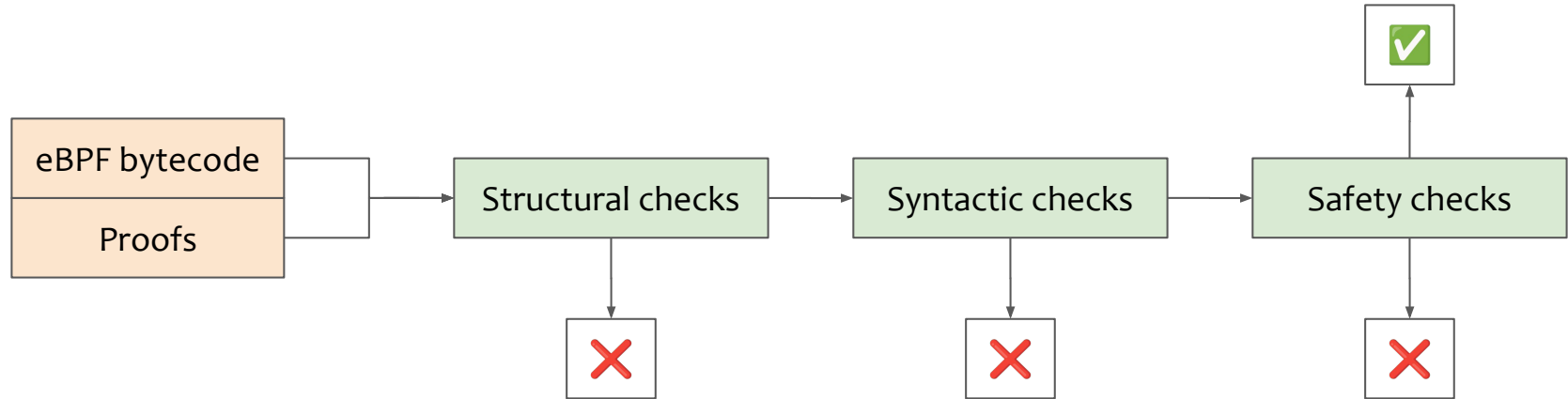
19: w0 = w6
20: exit
```

Re-establish
 $r2 = i \in [0, 7]$

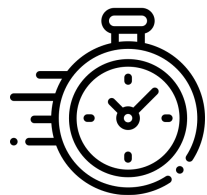


Untrusted Userspace: Iterative Verification

- Verification in layers
- Incorrect proofs are rejected early

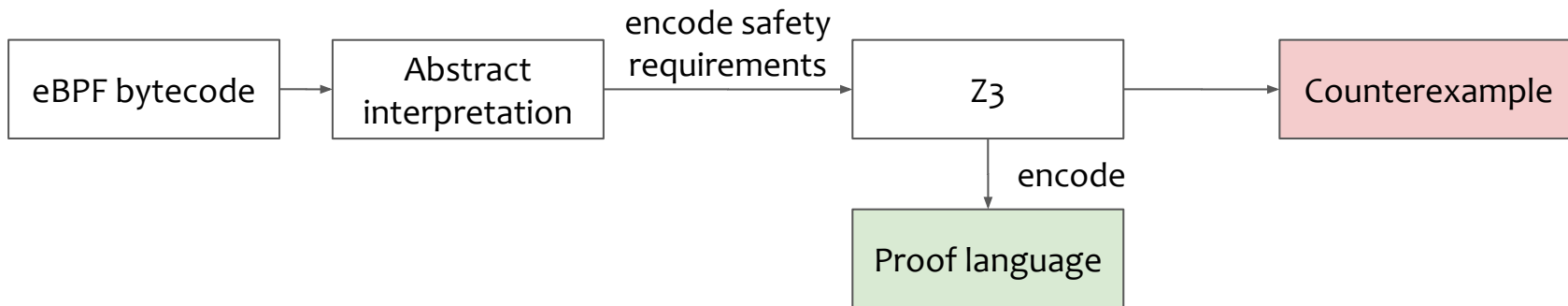


How can we generate proofs?



Userspace is **unconstrained**:

It may run expensive analyses, e.g. using SMT solvers



Generating Proofs: Abstract Interpretation

```
7: call 0x1 // bpf_map_lookup_elem(map0, &key)
```

```
abstract state at pc 8:  
  r0 = ptr(map_value(map0), maybe_null, off=0, size=64, perm=rw)
```

```
8: if r0 == 0 goto <pc 23>
```

```
edge pc 8 -> pc 9:  
  NarrowNonNull(r0 != 0)
```

```
abstract state at pc 8:  
  r0 = ptr(map_value(map0), nonnull, off=0, size=64, perm=r)
```

```
14: r3 = *(u64 *)(r0 + 0x0) // load eight bytes
```

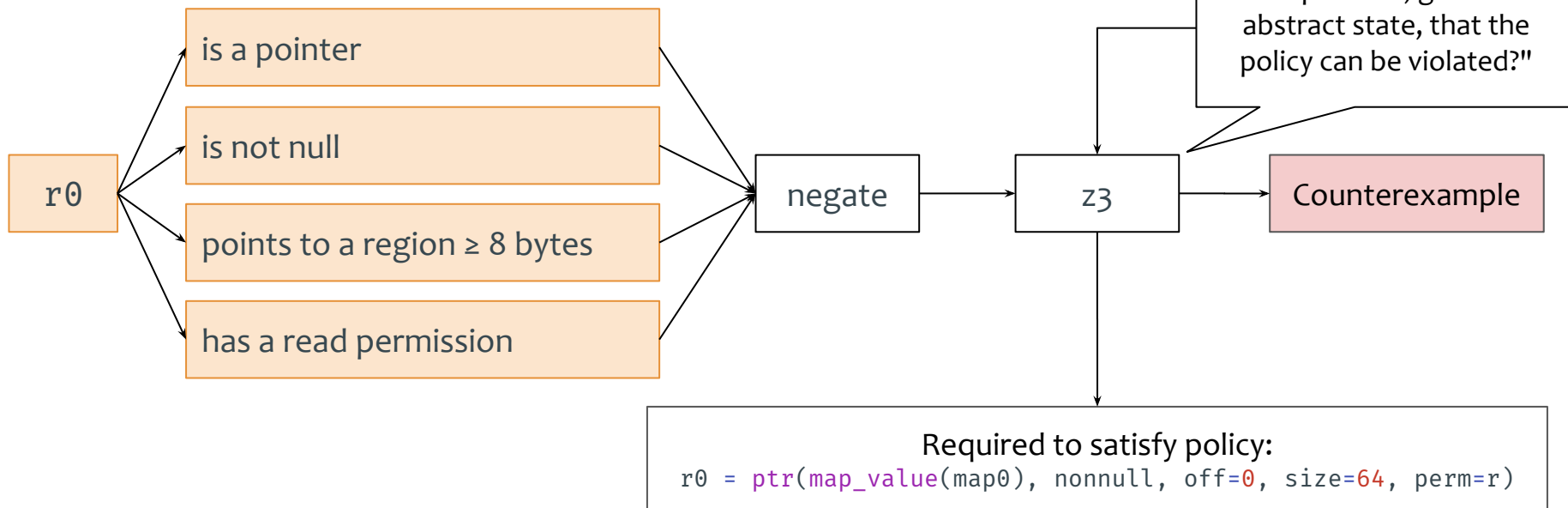
Policy:
`bpf_map_lookup_elem(map0, ...)`
returns a `maybe_null` pointer

NarrowNonNull: $p \neq 0$,
 $\text{Ptr}(p, \dots) \vdash \text{Ptr}(p,$
nonnull ...)

Generating Proofs: Encoding Safety Requirements

```
abstract state at pc 8:  
  r0 = ptr(map_value(map0), nonnull, off=0, size=64, perm=r)
```

```
8: r3 = *(u64 *)(r0 + 0) // load eight bytes
```



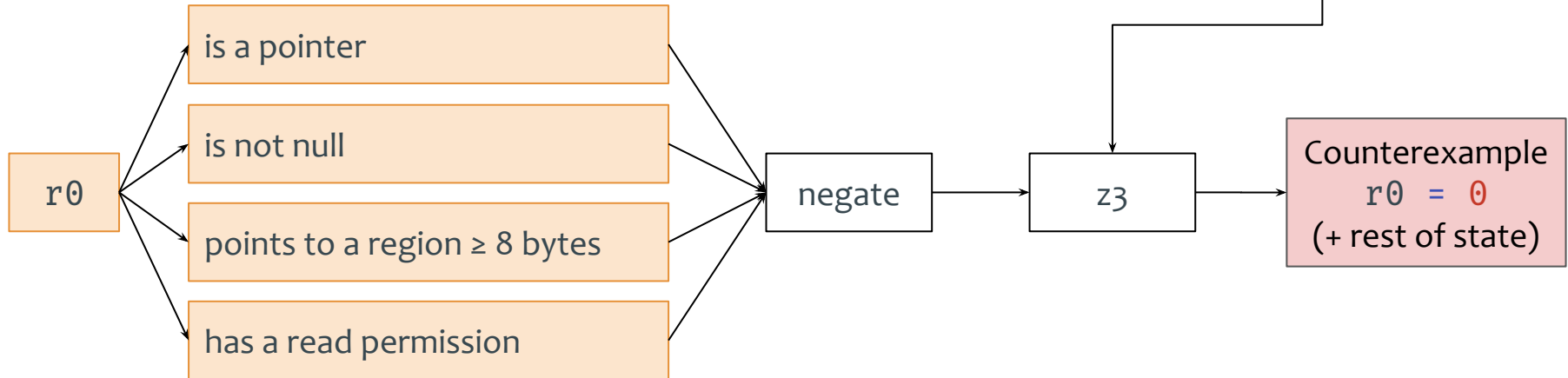
Example: Violating the Policy

no null check

abstract state at pc 8:

```
r0 = ptr(map_value(map0), maybe_null, off=0, size=64, perm=r)
```

```
14: r3 = *(u64 *)(r0 + 0) // load eight bytes
```



Generating Proofs: Loop Invariants

```
p = bpf_map_lookup_elem(&map, &key); // 64-byte map value
```

```
abstract state:
```

```
p = ptr(map_value(map0), maybe_null, off=0, size=64, perm=r)
```

```
if (!p || n > 8)
    return 0;
```

```
edge hints:
```

```
NarrowNonNull(p)
NarrowUpperFromLt(n, 8)
```

```
abstract state:
```

```
r0 = ptr(map_value(map0), nonnull, off=0, size=64, perm=r)
n ∈ [0, 7]
```

```
for (i = 0; i < n; i++)
```

```
edge hints:
```

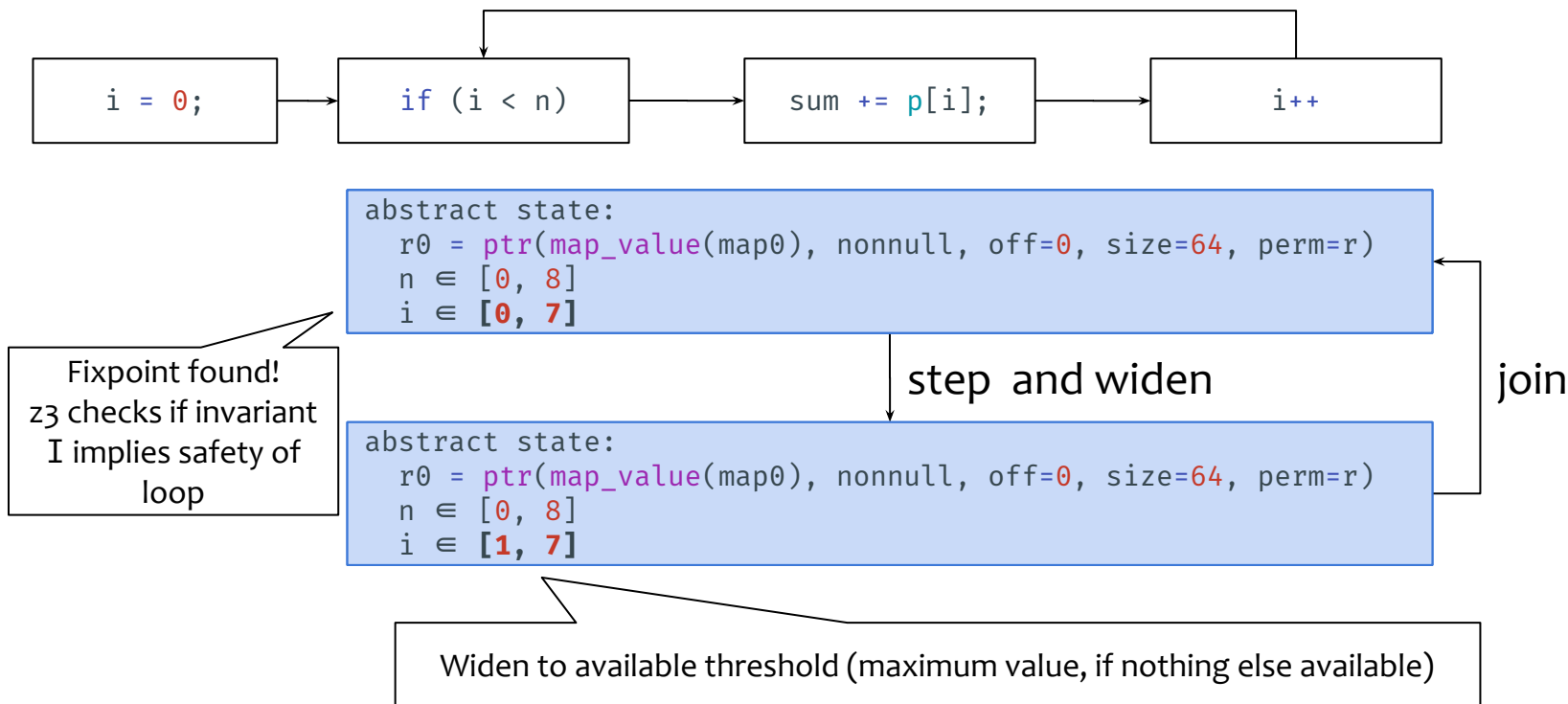
```
NarrowUpperFromLt(i, n)
```

```
sum += p[i];
```

Loop invariant:
 $i = ?$

Generating Proofs: Loop Invariants

Fixpoint iteration, until loop header invariant is found



Generating Proofs: Loop Termination

Synthesize a termination measure [10] after loop invariants have been found:

Invariant I

```
r0 = ptr(map_value(map0), nonnull, off=0, size=64, perm=r)
n ∈ [0, 8]
i ∈ [0, 7]
```

Transition

```
i = i + 1
```

Z3 solves

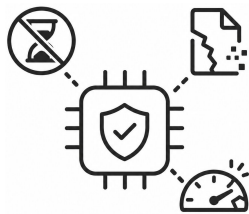
```
ρ(i, n, ...) = c0 + c1 * i + c2 * n + ... for all states
∀ i: I => ρ(i) >= 0                                ρ is always positive under invariant I
ρ(state') <= ρ(state)                                ρ never increases in a loop
ρ(state') < ρ(state) if back-edge ρ always decreases on back edges
```

Solution

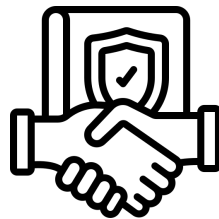
```
ρ(i, n, ...) = 8 + -1 * i + 0 * n = 7 - i
```

```
ρ' = 7 - (i + 1) = ρ - 1
```

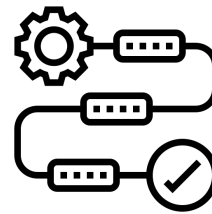
Challenges Applying PCC to eBPF



In-kernel verification



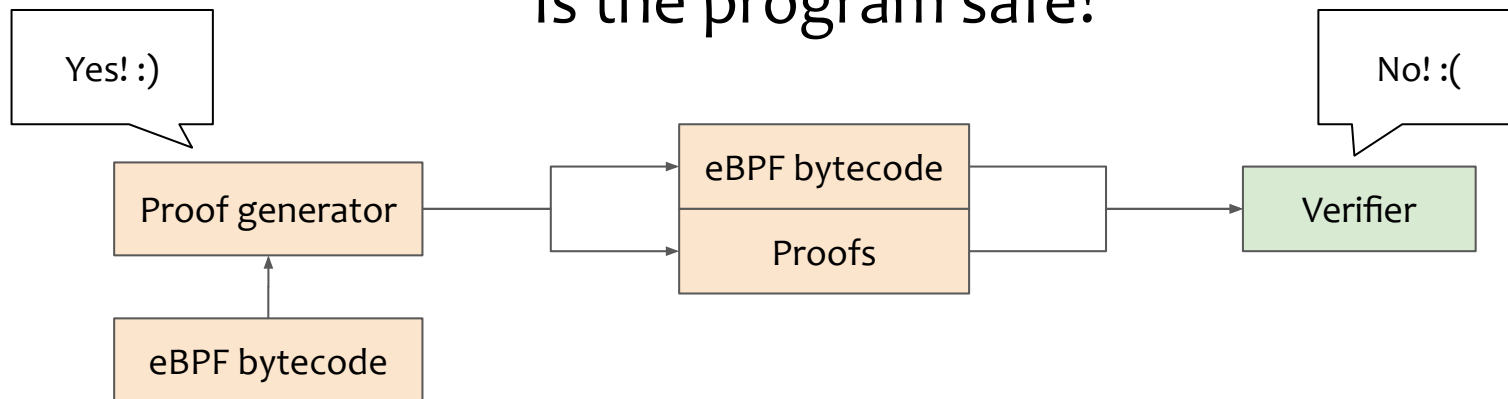
Proof generator and
verifier must agree



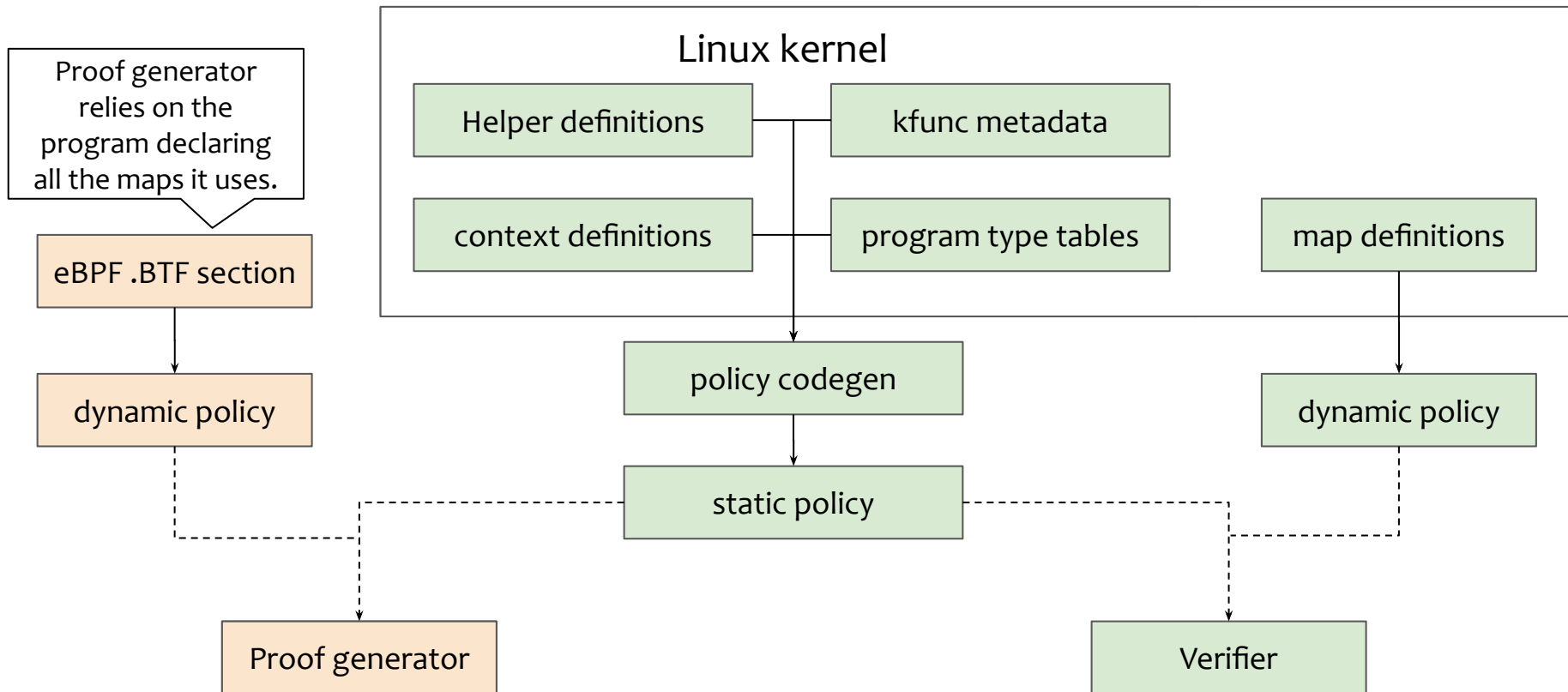
JIT must transform
proofs

Proof Generator and Verifier May Disagree

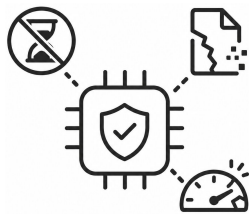
"Is the program safe?"



Proof generator and verifier need explicit shared rules for **instruction semantics, termination, memory accesses, and interface contracts.**



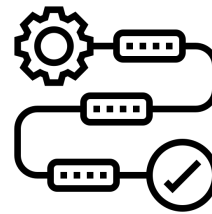
Challenges Applying PCC to eBPF



In-kernel verification



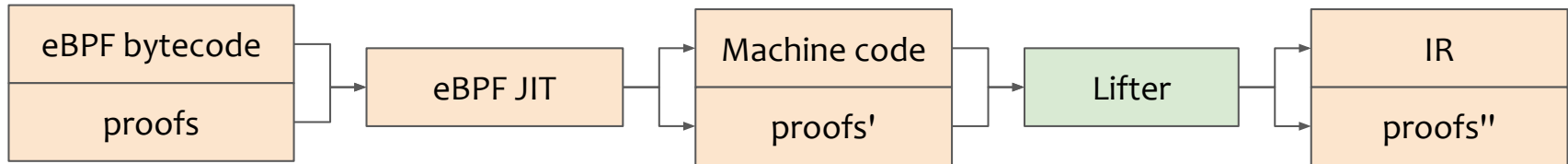
Proof generator and
verifier must agree



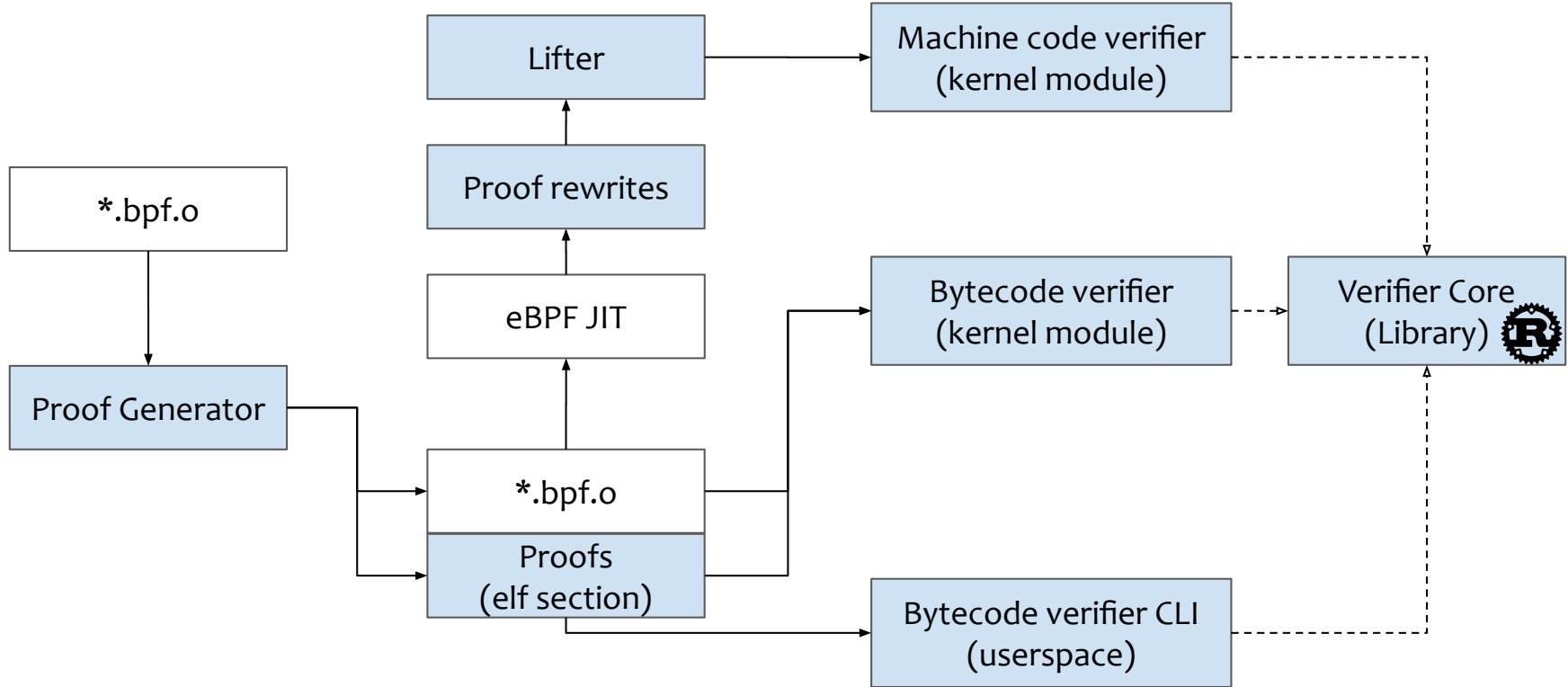
JIT must transform
proofs

Verification After JIT Compilation

- Proofs are generated on eBPF bytecode
- Define verification on an IR instead of bytecode directly
- While transforming code, JIT compilers transform proofs alongside it



Implementation: HiveGuard



Evaluation

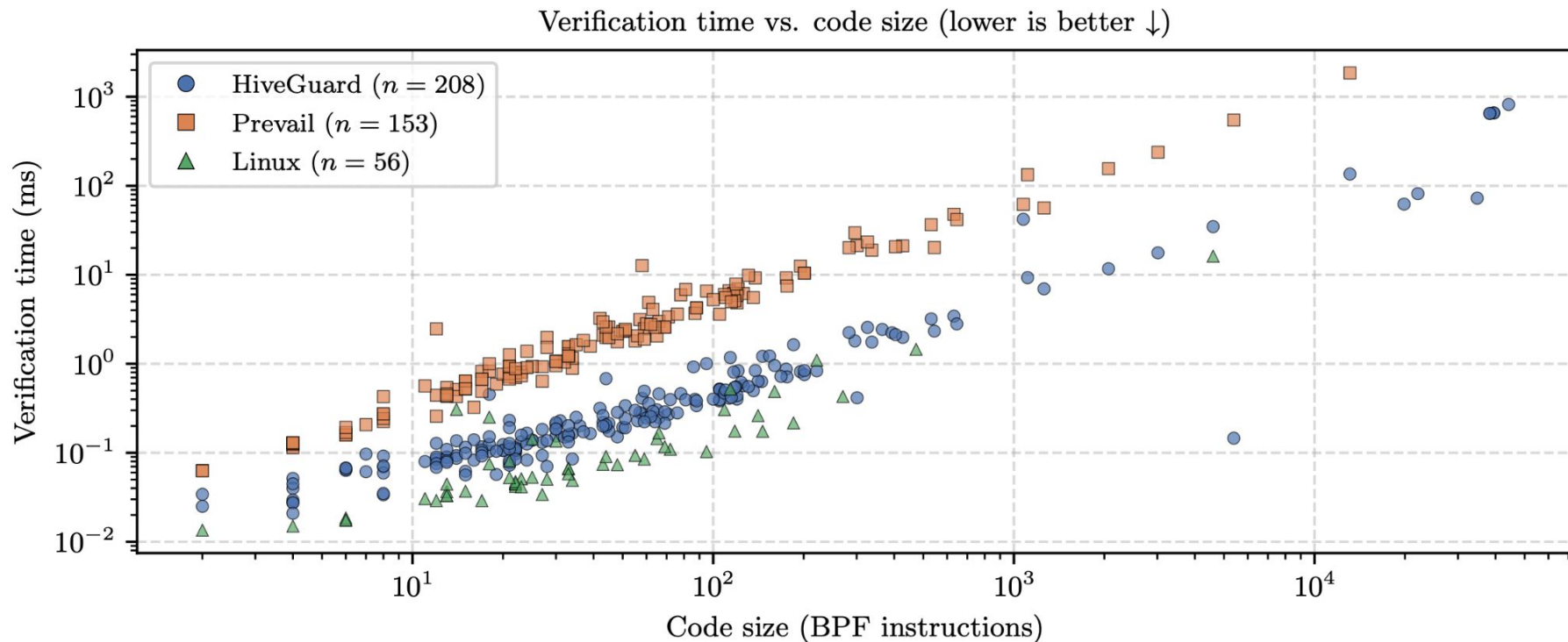
- **RQ1: Effectiveness**
Can HiveGuard verify real-world programs?
- **RQ2: Performance**
How fast is proof generation and verification?
- **RQ3: Trust**
Does HiveGuard reduce the trust in the kernel?

RQ1: Can HiveGuard verify real-world programs?

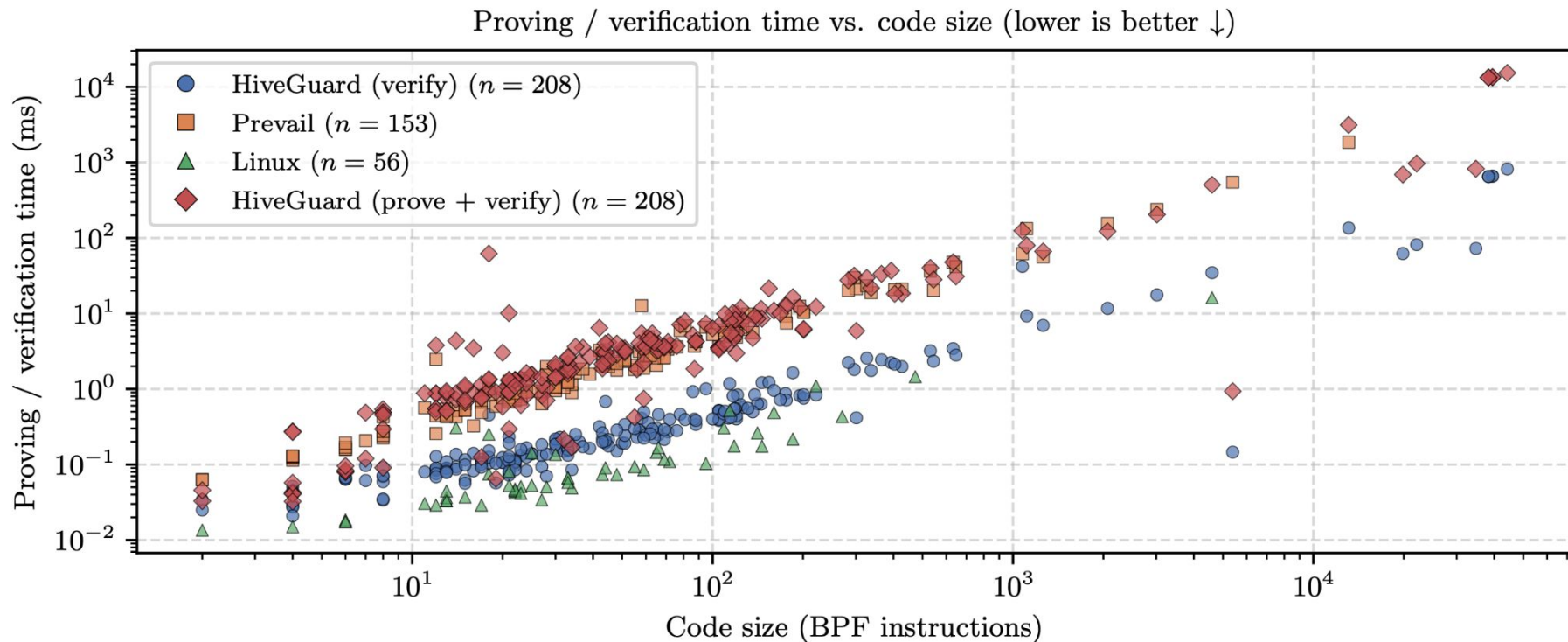
- eBPF Samples [5]: 244 real-world programs
 - linux, cilium, bcc, prevail self-tests
 - **216 valid and safe programs, containing code**
 - 156 programs that can be loaded into the kernel

Verifier	Accepted programs
Linux verifier (affected by loader rejections)	100/156
PreVail	153/216
HiveGuard (this work)	208/216

RQ2: How fast is proof verification?



RQ2: How fast is proof generation?



RQ3: Does HiveGuard reduce the trust in the kernel?

HiveGuard

Component	SLoC
Policy tables (auto-generated)	8.5k
Abstract transfer, safety checks	7.3k
Post-JIT verification	2.5k
IR	2.3k
Proof format (parse, decode, validate)	2.2k
Lemma catalogue	1.6k
Termination checking	1k
Other	3.9k
Total	29.3k

Linux

Component	SLoC
verifier.c	17.8k
BTF parsing + type validation	6.5k
Context access rules	5.9k
Helper prototype tables	2.2k
JIT (arm64)	2.5k
Total	34.9k

Verifier	Approach	Main distinction
Exoverifier (LPC'21) [11]	Whole-program safety exported from Lean formalization	General theorem prover
VEP (NSDI'25) [12]	Annotated C, userspace verifier, bytecode proof checker	Source annotations, modified compiler
BCF (SOSP'25) [13]	Userspace on-demand proof refinement	Retains existing verifier, calls into userspace
Proof-Carrying eBPF (eBPF'26) [14]	Augment existing verifier, generate BCF proofs ahead of time	Retains existing verifier + BCF proofs
HiveGuard (LPC'26) (this work)	Abstract interpretation, SMT-generated proofs, linear ranking functions, lemma hints, termination rankings	Replaces in-kernel verification with bounded certificate checking

[11]: Luke Nelson et. al. "A proof-carrying approach to building correct and flexible BPF verifiers". Linux Plumbers Conference 2021

[12]: Xiwei Wu et. al. "VEP: a two-stage verification tool chain for full eBPF programmability". USENIX NSDI '25

[13]: Hao Sun et. al. "Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement". SOSP '25

[14]: Yalu Cai et. al. "Proof-Carrying-BPF Programs for the eBPF Verifier". eBPF '26

Summary: Proof-Carrying eBPF Verification

How can we build a verifier architecture that decouples complex safety reasoning from the trusted mechanism that enforces its results?

We introduce **HiveGuard**, which splits expensive proof searching and proof verification by applying Proof-Carrying Code.

HiveGuard shows that Proof-Carrying Code can be applied to eBPF kernel verification.



Personal website

Questions?

<https://finkmartin.com> ♦ martin.fink@cit.tum.de

Backup

- Resource tokens are minted at call-site based on eBPF instruction pointer
- Tokens are linear types:
 - Must be released **exactly once**: freed, returned to the caller, ...
- Policy defines what helper functions mint tokens/release them
- After release, checker invalidates all copies of the token in registers/memory assertions

```
Invariant ::= Atom1 ∧ ... ∧ Atomn
            | Invariant1 ∨ ... ∨ Invariantn
at block entry
```

```
Term ::= r
        | ghost
        | constant
        | r + constant
        | ghost + constant
        | r1 - r2
```

```
Atom ::=
    AbstractValue(term, state)
    | StackSlot(frame, [lo, hi], state)
    | MemoryCell(region, [lo, hi], state)
    | Eq(lhs, rhs)
    | Neq(lhs, rhs)
    | NeqZero(term)
    | Lt(lhs, rhs, signed)
    | Le(lhs, rhs, signed)
    | Ghost(ghost, term)
    | Def(dst, op, lhs, rhs)
    | Owns(token, resource)
    | PacketLength(term)
    | LinearInduction(ptr, index, stride, base)
```

- Comparisons and ranges
 - NarrowUpperFromLt
 - NarrowUpperFromLe
 - NarrowLowerFromLt
 - NarrowLowerFromLe
 - RangeIntersect
 - RangeFromEq
 - SignRange
 - NeqCongruenceNarrow
- Pointers and equality
 - NarrowNonNull
 - NeqZeroFromEq
- Integer operations
 - MaskBoundConst
 - ShiftBoundConst
 - MulConstBound
 - XorBoundConst
- Packets
 - PacketBoundFromCmp
 - PacketSizeShares
 - PacketLengthFromDiff
 - PacketBoundFromLength
- Pointer induction
 - LinInductionShift
 - LinInductionBound

Why Multiple Disjunctions?

- Prevents losing information when joining after correlated branches
- Bounded number of disjunctions: Block must hold for all of them, satisfy one of each successor

```
p = map_lookup_elem(map1, ...);
q = map_lookup_elem(map2, ...);

if (something) {
    if (p == NULL) return;
    tag = 1;
} else {
    if (q == NULL) return;
    tag = 2;
}

/* join block:
    tag == 1 && p != NULL
**OR** tag == 2 && q != NULL
*/
if (tag == 1)
    *p = 0;
else
    *q = 0;
```

Policy Example



```
policy BPF_PROG_TYPE_XDP {
  entry {
    r1 = ptr(ctx, nonnull, off=[0, 0], size=24, align=1, perm=r)
    r10 = ptr(stack(frame0), nonnull, off=[0, 0], size=512, align=8, perm=rw)
    stack_size = 512
    ctx[0, 4) = ptr(packet, nonnull, off=[0, 0], size=0, align=1, perm=rw)
    ctx[4, 8) = ptr(packet_end, nonnull, off=[0, 0], size=0, align=1, perm=r)
    ctx[8, 12) = ptr(packet_meta, nonnull, off=[0, 0], size=0, align=1, perm=rw)
  }
  helpers {
    0x01: ret = bpf_map_lookup_elem(v0, v1)
    where map$0 in {HASH, ARRAY, PERCPU_HASH, ...}
    v0: ptr(map(map$0), nonnull, off=[0, 0], size=0, align=8, perm=r)
    ret: ptr(map_value(map$0, uid=0), maybe_null, off=[0, 0], size=16, align=8, perm=rw)
    0x59: ret = bpf_map_peek_elem(v0, v1)
    where map$0 in {QUEUE, STACK, BLOOM_FILTER}
    v0: ptr(map(map$0), nonnull, off=[0, 0], size=0, align=8, perm=r)
    effect init_mem v1 size=value_size(v0)
    ...
  }
}
```