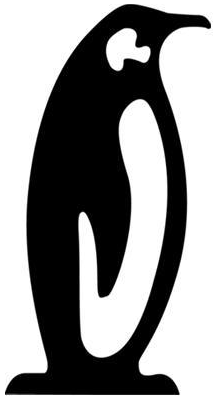


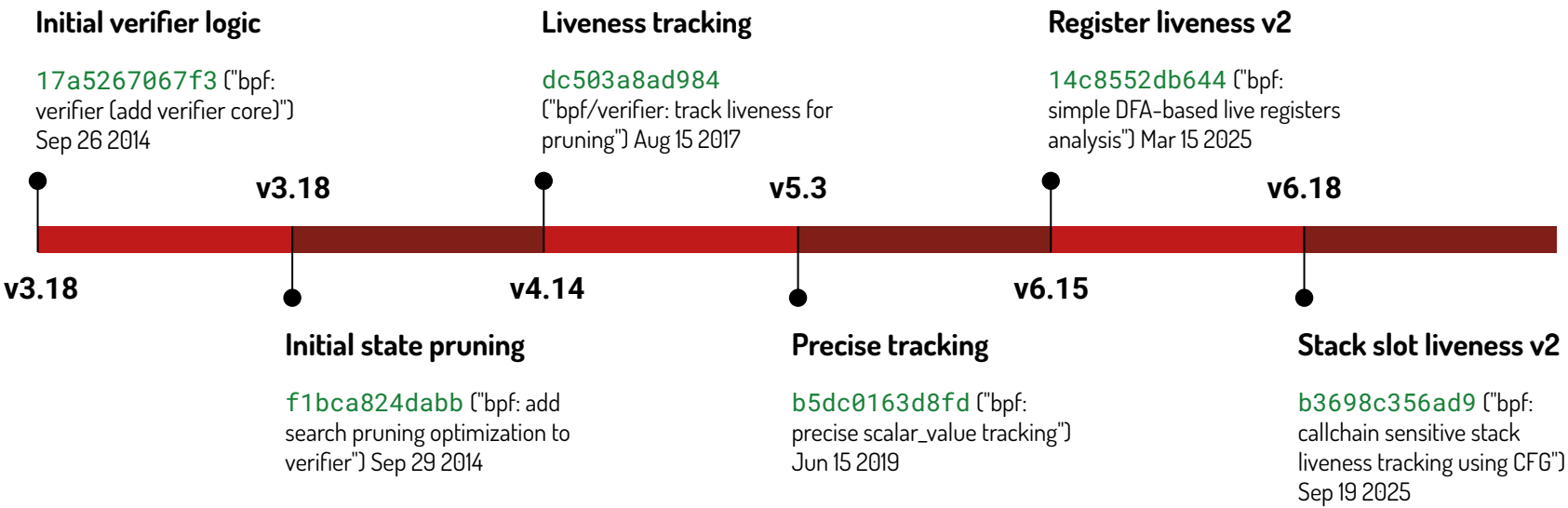
東京 2025 LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

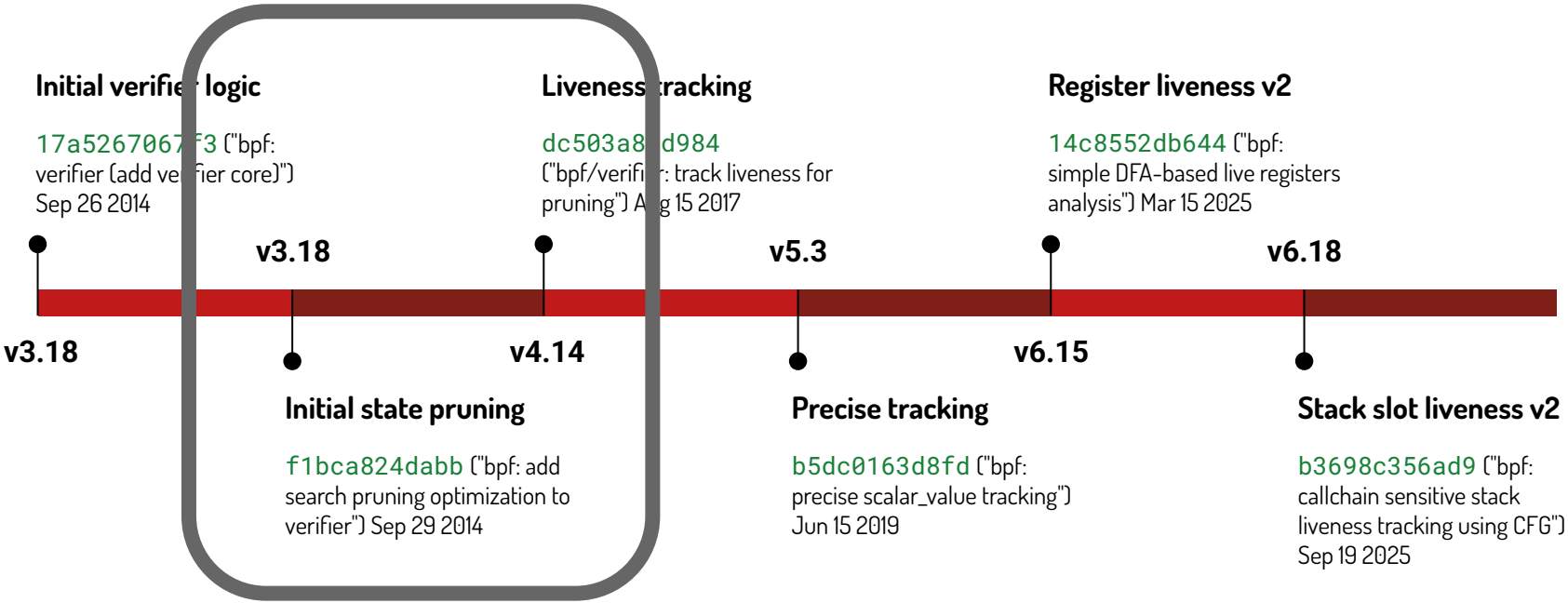
Making Sense of State Pruning

東京 2025
LINUX
PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

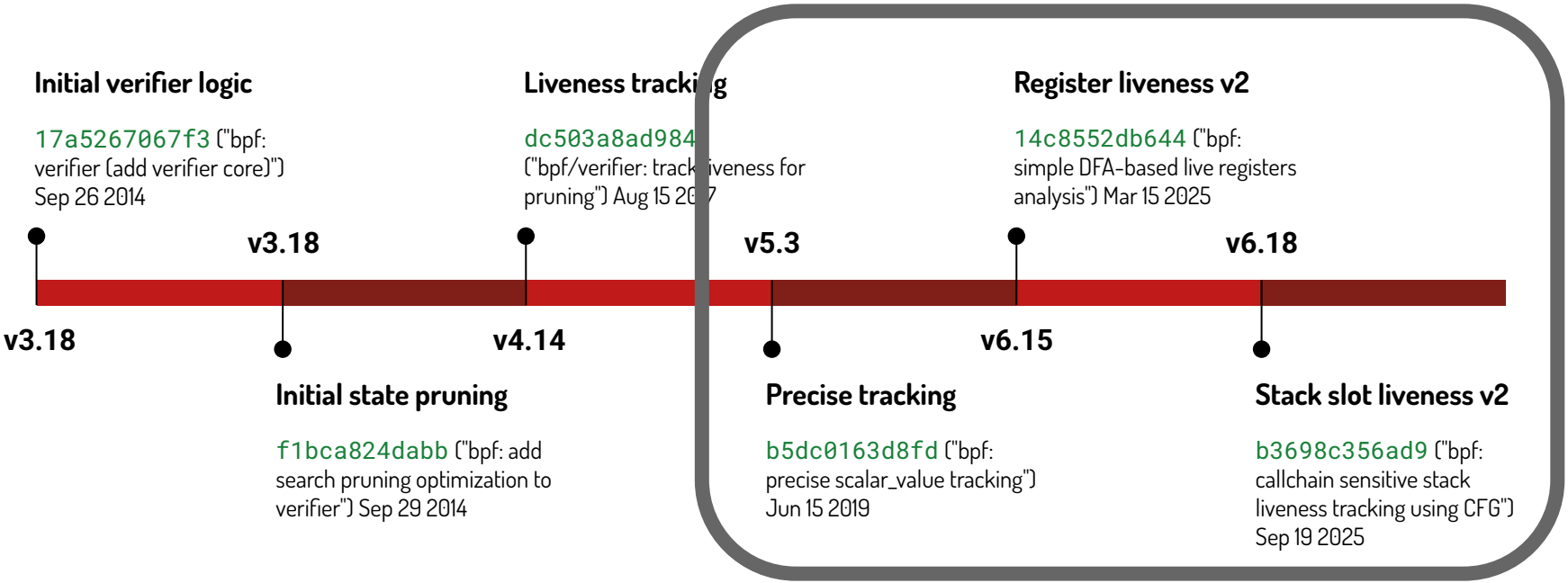
Timeline


Initial verifier logic
17a5267067f3 ("bpf: verifier (add verifier core)") Sep 26 2014
Liveness tracking
dc503a8ad984 ("bpf/verifier: track liveness for pruning") Aug 15 2017
Register liveness v2
14c8552db644 ("bpf: simple DFA-based live registers analysis") Mar 15 2025
v3.18
v3.18
v4.14
v5.3
v6.15
v6.18
Initial state pruning
f1bca824dabb ("bpf: add search pruning optimization to verifier") Sep 29 2014
Precise tracking
b5dc0163d8fd ("bpf: precise scalar_value tracking") Jun 15 2019
Stack slot liveness v2
b3698c356ad9 ("bpf: callchain sensitive stack liveness tracking using CFG") Sep 19 2025

Timeline


Initial verifier logic
17a5267067f3 ("bpf: verifier (add verifier core)") Sep 26 2014
Liveness tracking
dc503a8d984 ("bpf/verifier: track liveness for pruning") Aug 15 2017
Register liveness v2
14c8552db644 ("bpf: simple DFA-based live registers analysis") Mar 15 2025
v3.18
v3.18
v4.14
v5.3
v6.15
v6.18
Initial state pruning
f1bca824dabb ("bpf: add search pruning optimization to verifier") Sep 29 2014
Precise tracking
b5dc0163d8fd ("bpf: precise scalar_value tracking") Jun 15 2019
Stack slot liveness v2
b3698c356ad9 ("bpf: callchain sensitive stack liveness tracking using CFG") Sep 19 2025

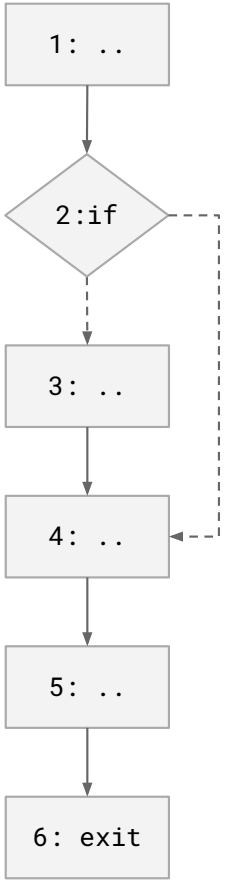
Timeline


Initial verifier logic
17a5267067f3 ("bpf: verifier (add verifier core)") Sep 26 2014
Liveness tracking
dc503a8ad984 ("bpf/verifier: track liveness for pruning") Aug 15 2017
Register liveness v2
14c8552db644 ("bpf: simple DFA-based live registers analysis") Mar 15 2025
v3.18
v3.18
v4.14
v5.3
v6.15
v6.18
Initial state pruning
f1bca824dabb ("bpf: add search pruning optimization to verifier") Sep 29 2014
Precise tracking
b5dc0163d8fd ("bpf: precise scalar_value tracking") Jun 15 2019
Stack slot liveness v2
b3698c356ad9 ("bpf: callchain sensitive stack liveness tracking using CFG") Sep 19 2025

Path explosion

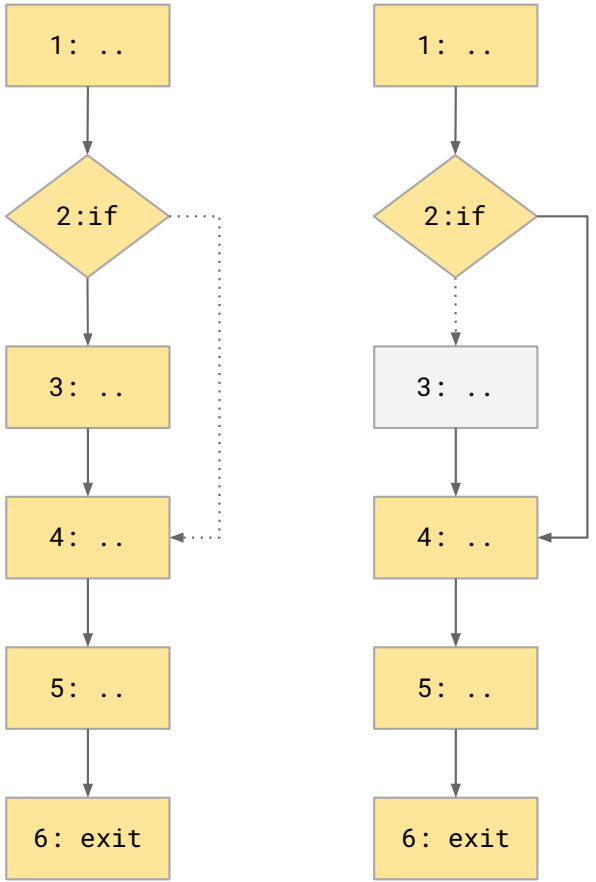
Path explosion is a fundamental problem that limits the scalability of program analyses.

The number of control-flow paths in a program grows exponentially with an increase in program size.



Path explosion

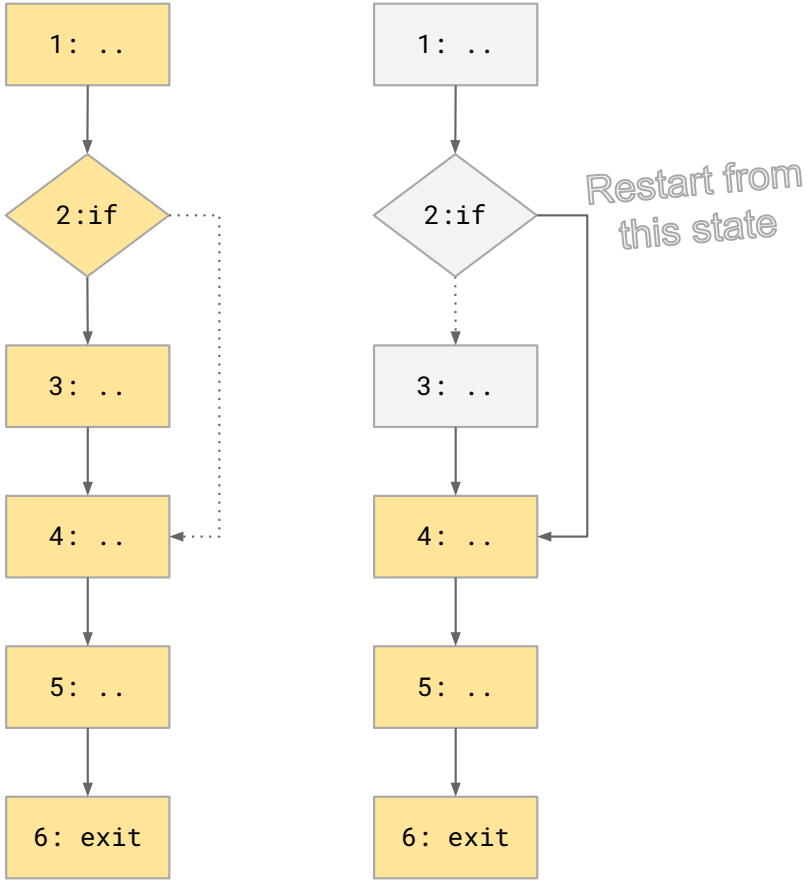
For example, a conditional jump creates an alternative control-flow in the program simulation.



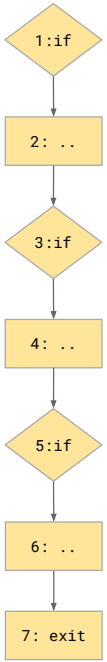
Path explosion

For example, a conditional jump creates an alternative control-flow in the program simulation.

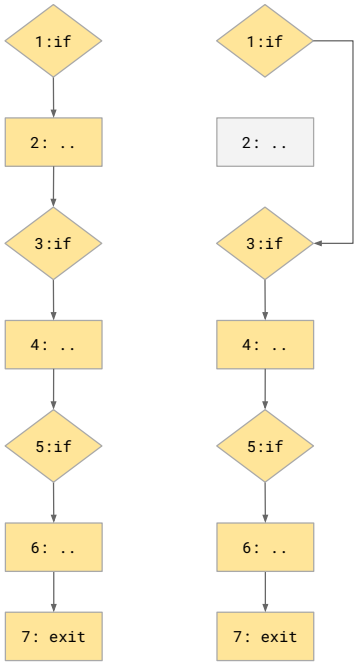
From 6 instructions, you now need to simulate 9.



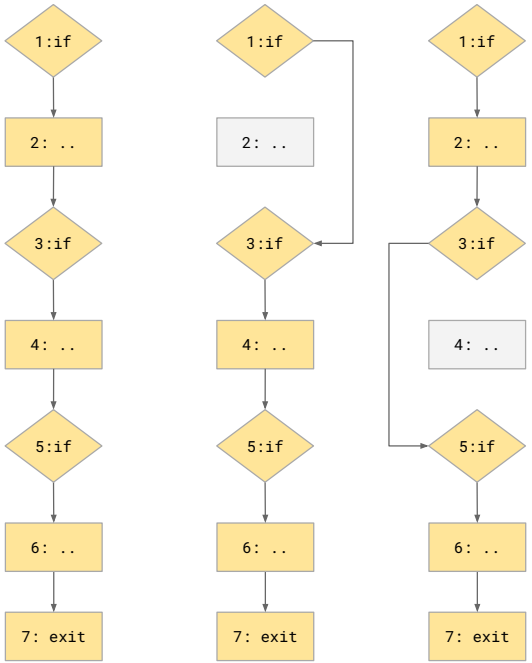
Path explosion



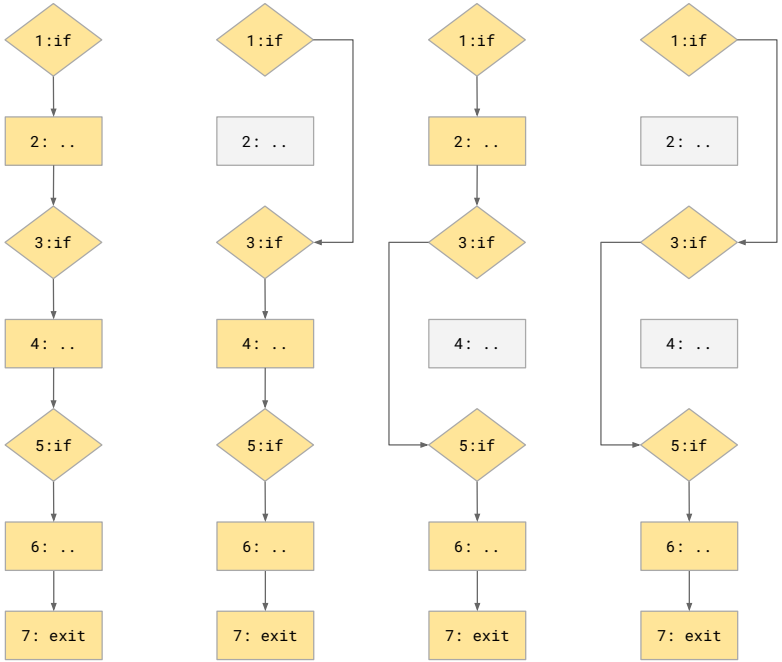
Path explosion



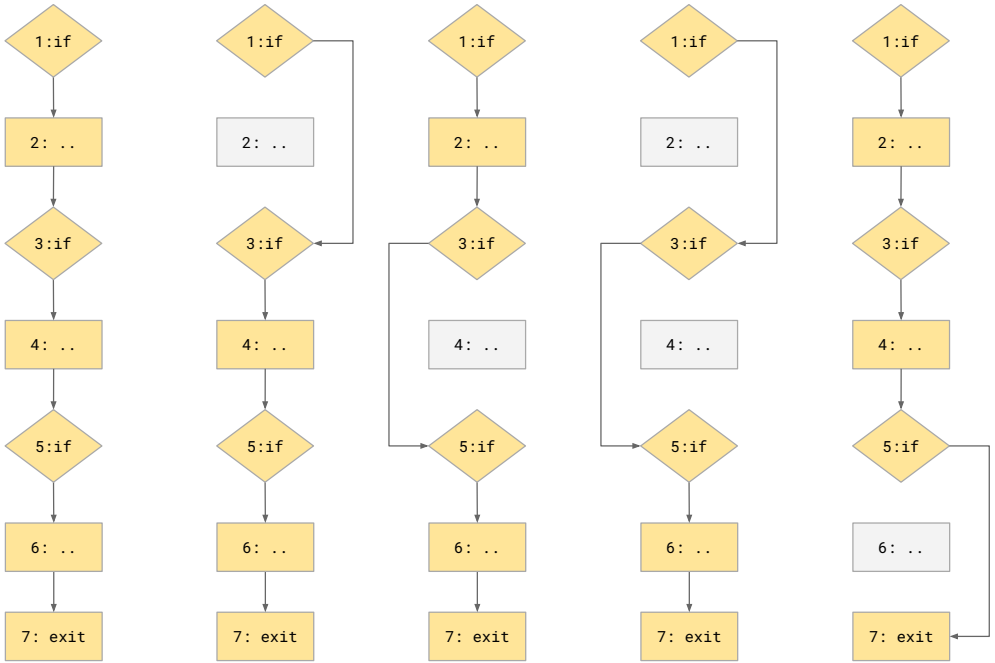
Path explosion



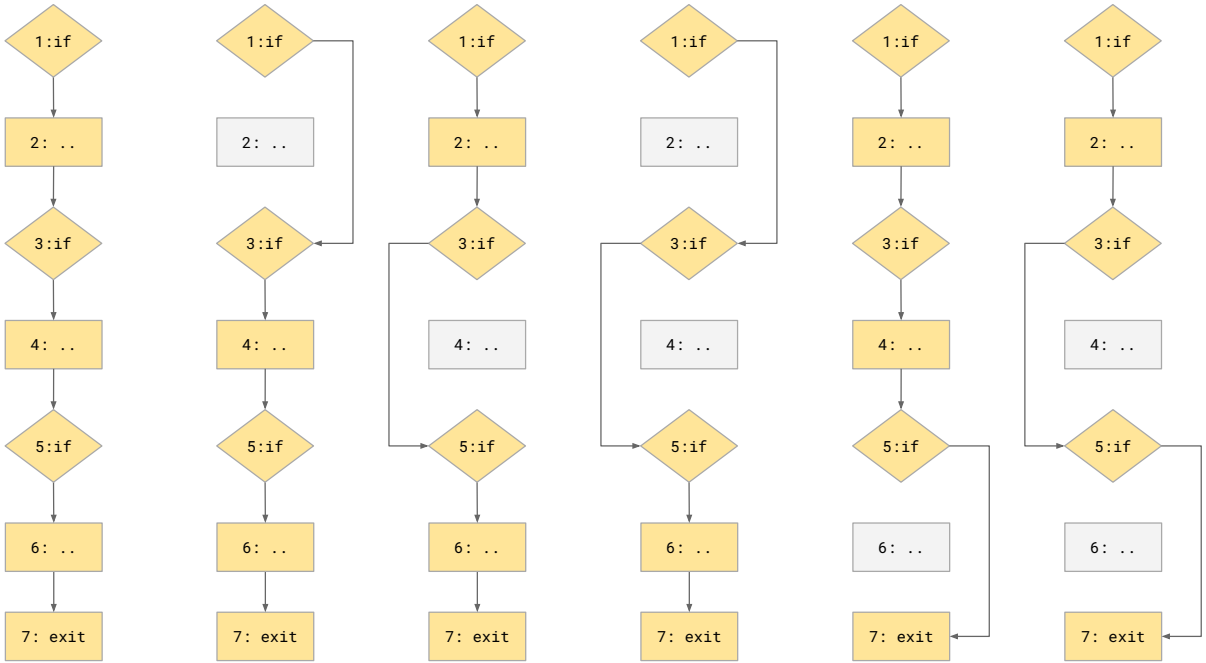
Path explosion



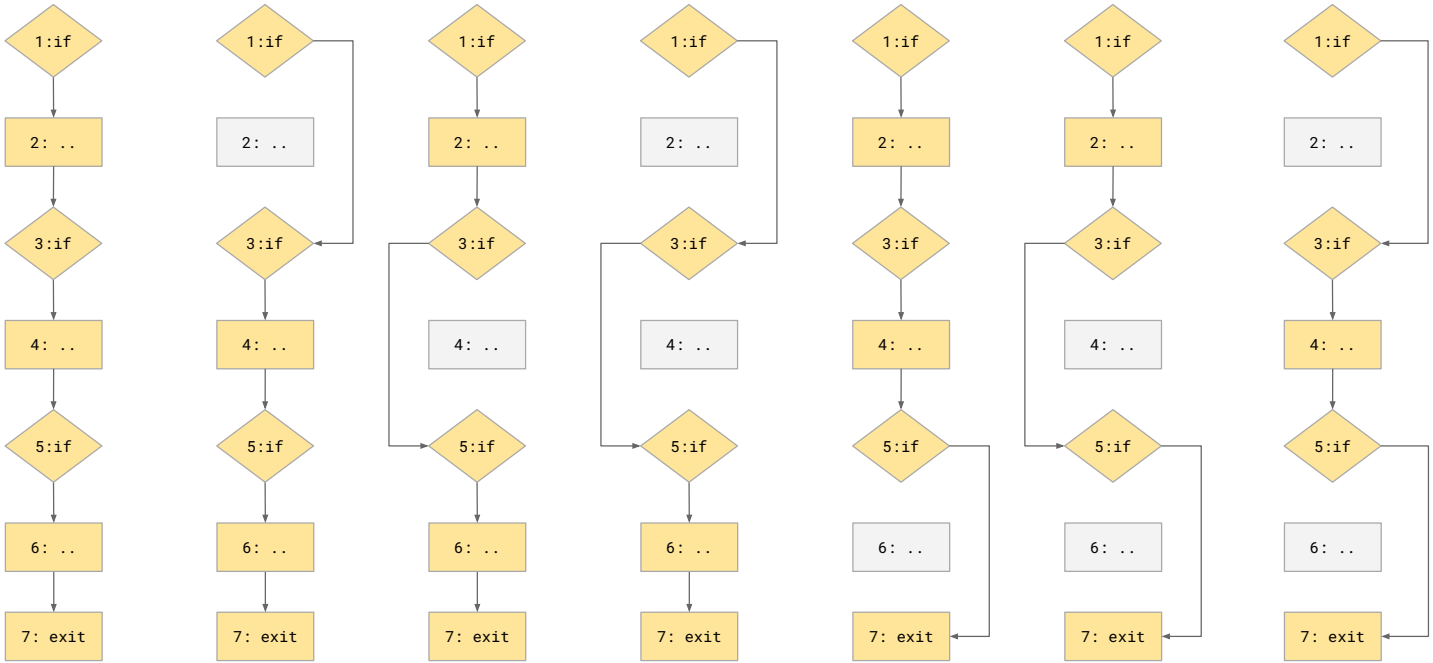
Path explosion



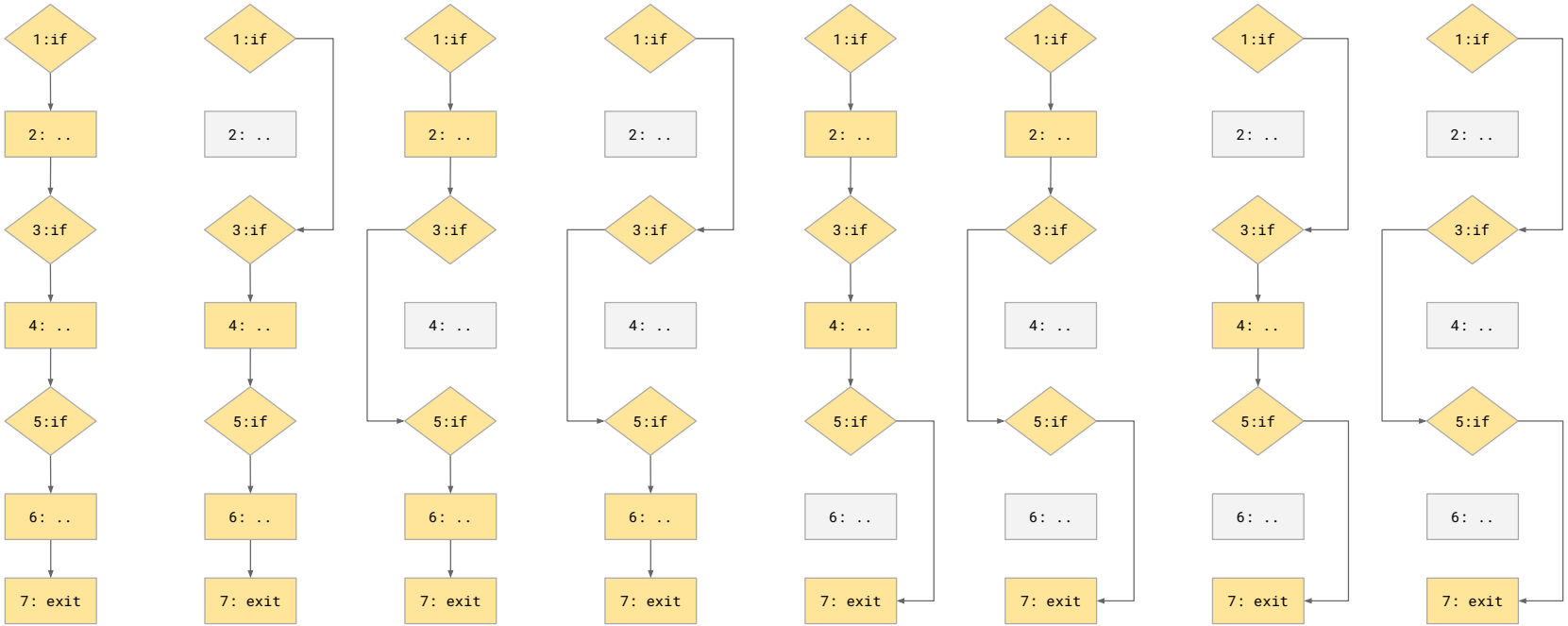
Path explosion



Path explosion



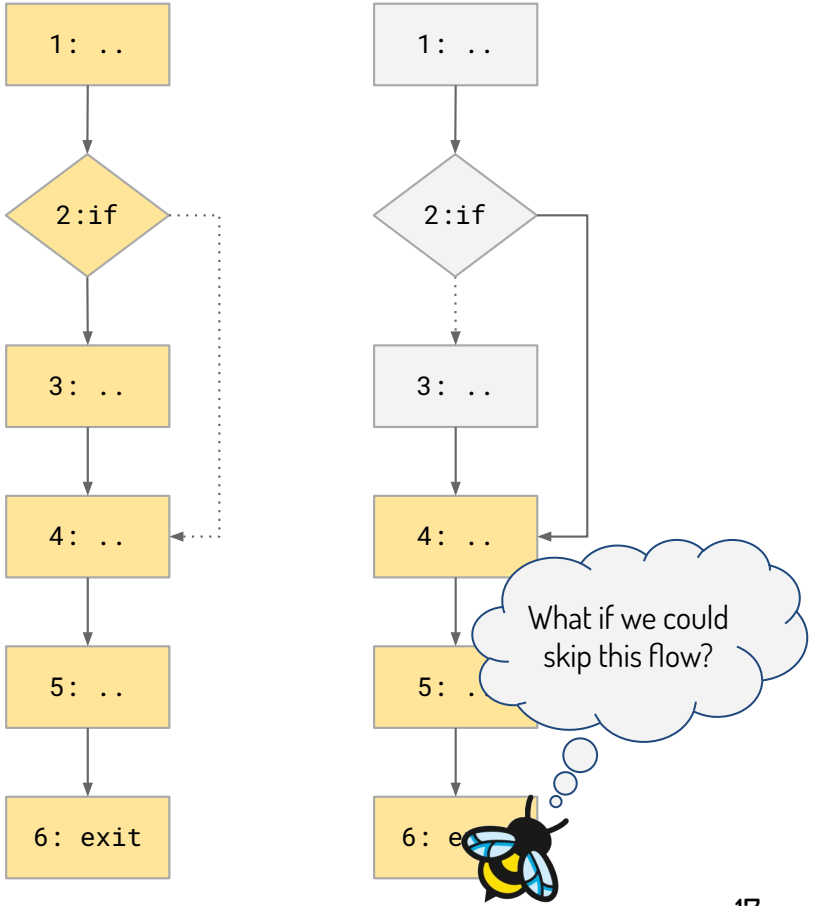
Path explosion



Path explosion

For example, a conditional jump creates an alternative control-flow in the program simulation.

From 6 instructions, you now need to simulate 9.

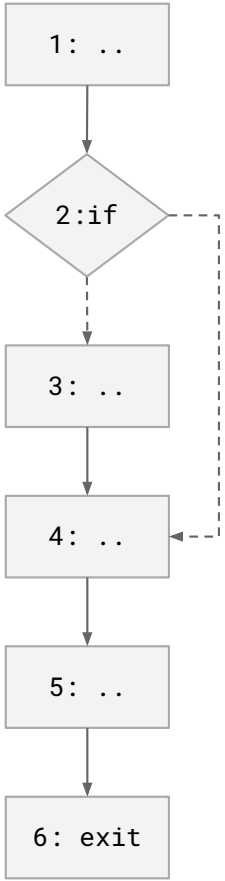


State pruning

Simple example taken from the first state pruning patch: `f1bca824dabb` ("bpf: add search pruning optimization to verifier").

The following example is simplified considering the state of state pruning from kernel v3.18.

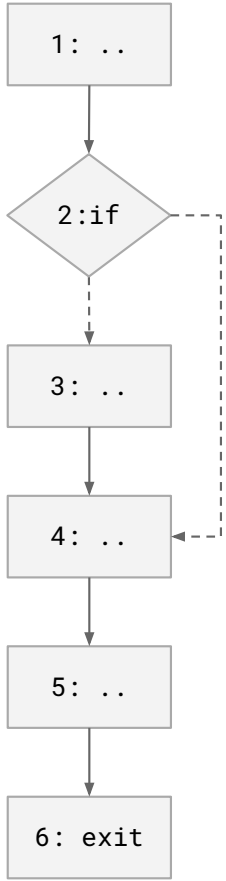
```
1: ..
2: if .. goto 4
3: ..
4: ..
5: ..
6: exit
```



State pruning: prune points

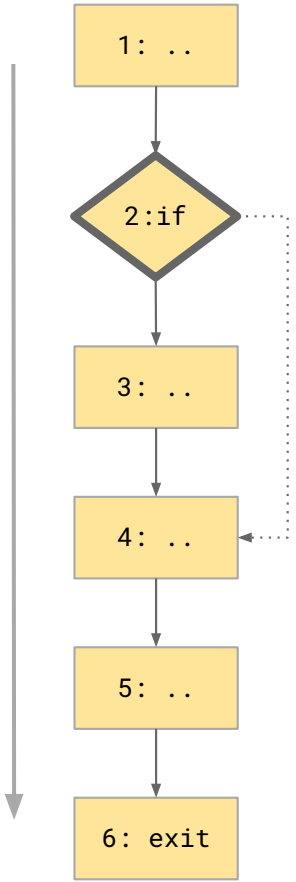
Pruning points are marking instructions on which the verifier will trigger state pruning.

On those instructions, states will be saved and equivalence will be checked.



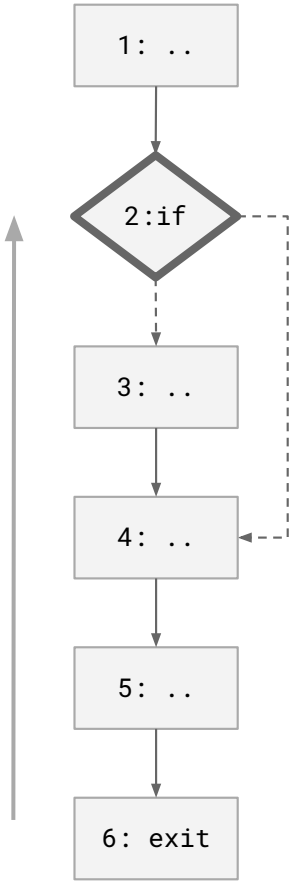
State pruning: prune points

The first pass is the **control flow graph check** (`check_cfg`) will first walk the prog, using depth-first search, to mark pruning points.



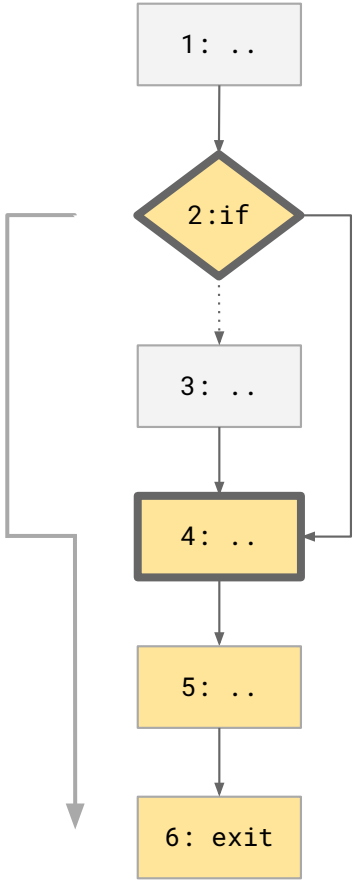
State pruning: prune points

The first pass is the **control flow graph check** (`check_cfg`) will first walk the prog, using depth-first search, to mark pruning points.

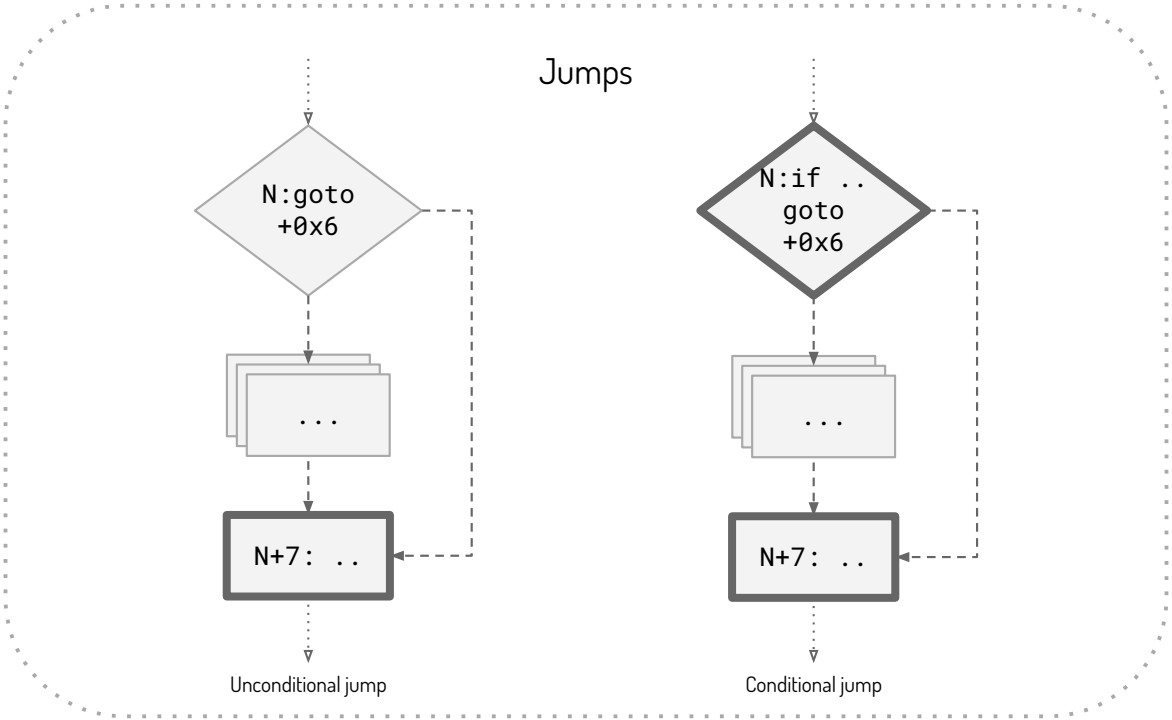


State pruning: prune points

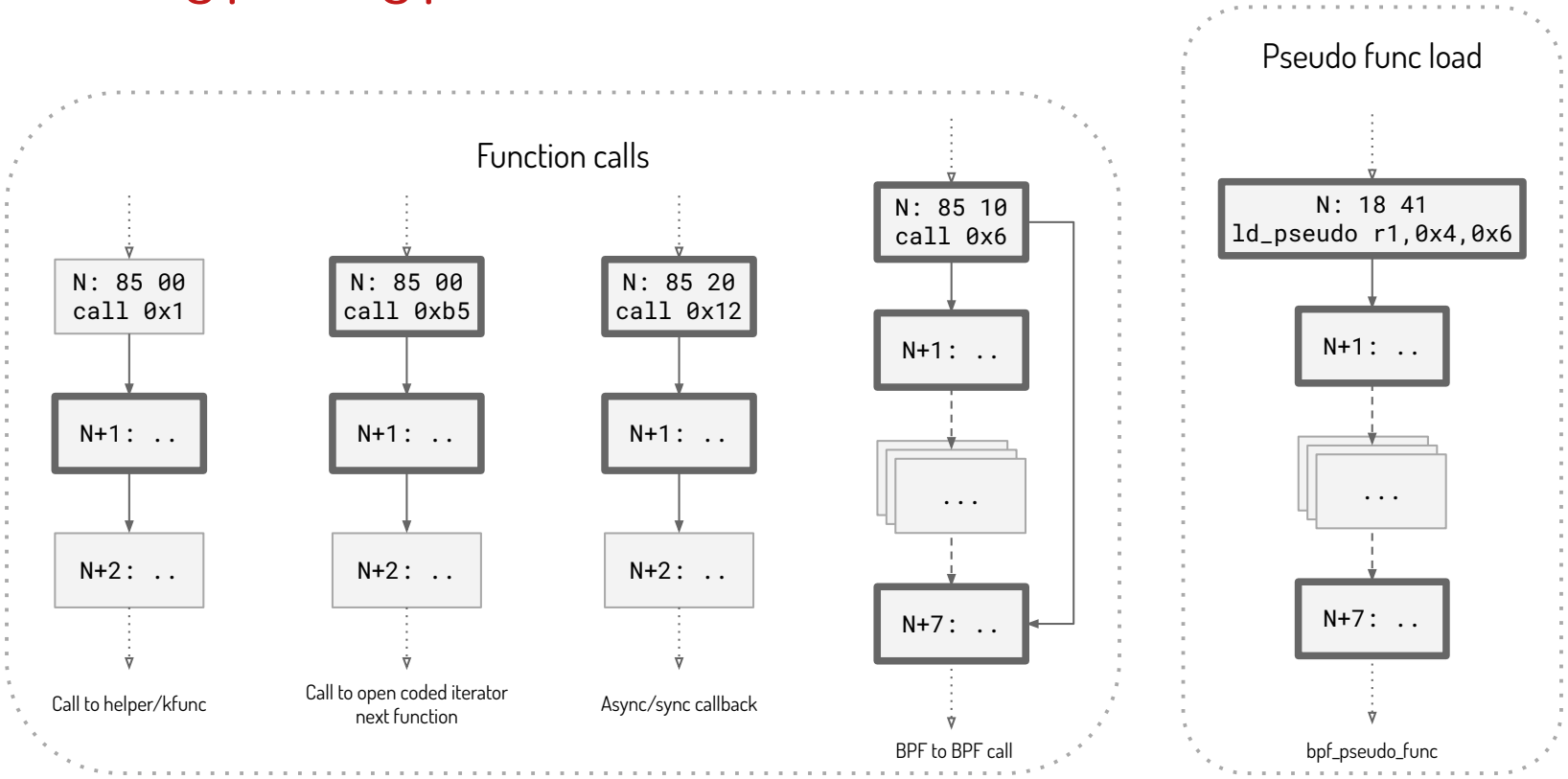
The first pass is the **control flow graph check** (`check_cfg`) will first walk the prog, using depth-first search, to mark pruning points.



Marking pruning points: jumps



Marking pruning points: calls



Marking pruning points: example

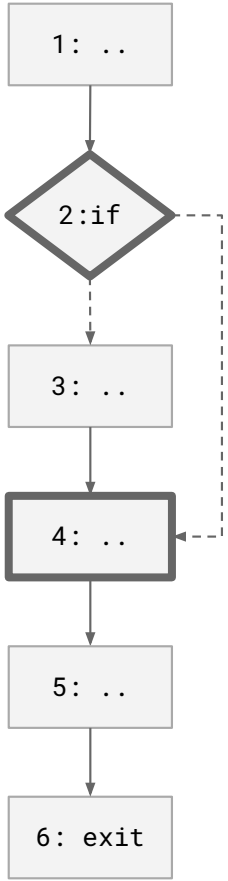
```
Live regs before insn and pruning points:
      0: ..........   (b7) r1 = 0
      1: .1........   (63) *(u32 *)(r10 -4) = r1
      2: ..........   (bf) r2 = r10
      3: ..2.......   (07) r2 += -4
      4: ..2.......   (18) r1 = 0xffff88c2853a9800
      6: .12.......   (85) call bpf_map_lookup_elem#1
      7: 0..345.... p (bf) r6 = r0
      8: ...3456... p (15) if r6 == 0x0 goto pc+6
      9: ...3456...   (b7) r1 = 4
     10: .1.3456...   (b7) r2 = 8
     11: .123456... p (85) call pc+5
     12: 0.....6... p (67) r0 <<= 32
     13: 0.....6...   (c7) r0 s>>= 32
     14: 0.....6...   (7b) *(u64 *)(r6 +0) = r0
     15: .......... p (b7) r0 = 0
     16: 0.........   (95) exit
     17: .12....... p (bf) r0 = r2
     18: 01........   (0f) r0 += r1
     19: 0.........   (95) exit
```

Patch from Mahé & Paul to print pruning points along live registers prints

State pruning: prune points

This example has 2 pruning points:

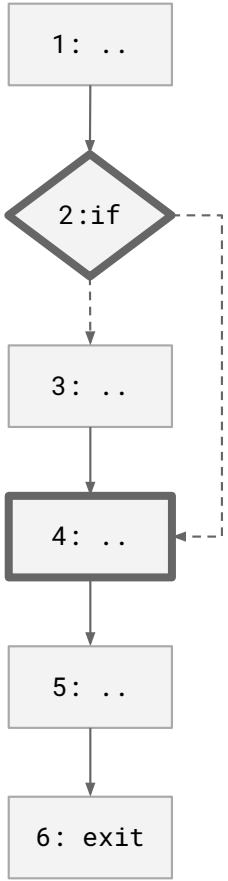
- one on the conditional jump instruction
- one at the jump landing instruction



State pruning: simulate execution

The second pass is `do_check`, simulating the execution of the program.

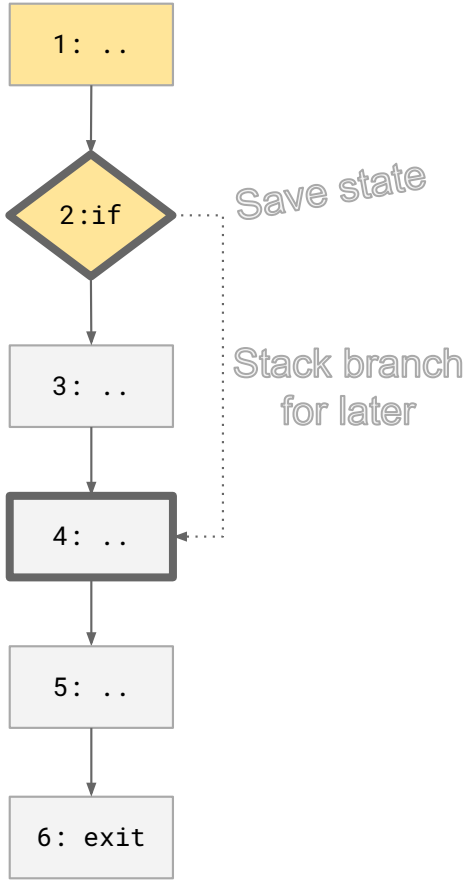
On each pruning points, the state pruning function is called.



State pruning: simulate execution

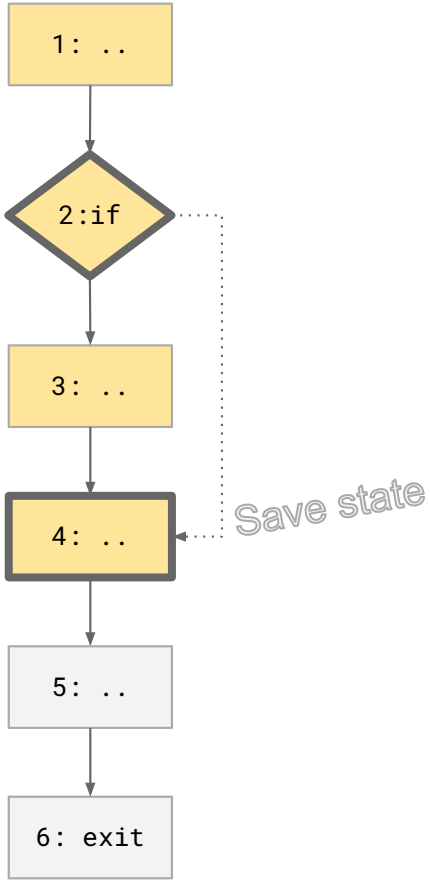
Arriving on instruction 2, the state pruning code is triggered and the state is saved.

For simulating taking the branch, the verifier will push the alternative branch path to the stack for later todo verification.



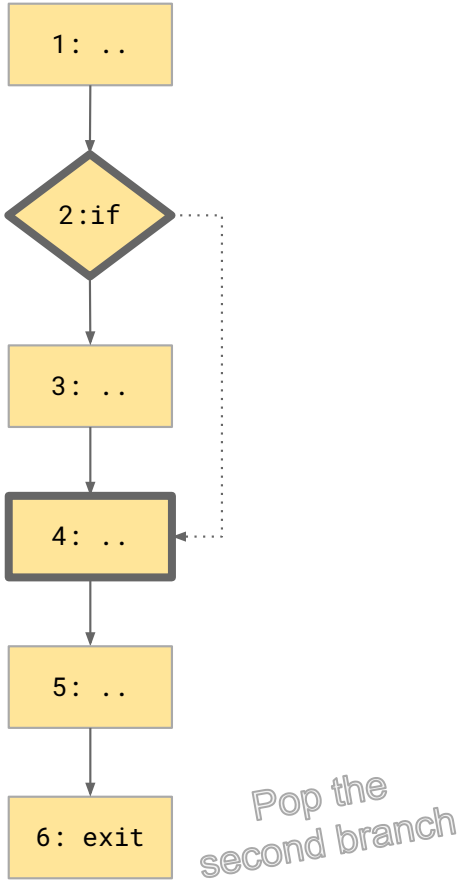
State pruning: simulate execution

Arriving on instruction 4, the state pruning code is triggered and the state is saved.



State pruning: simulate execution

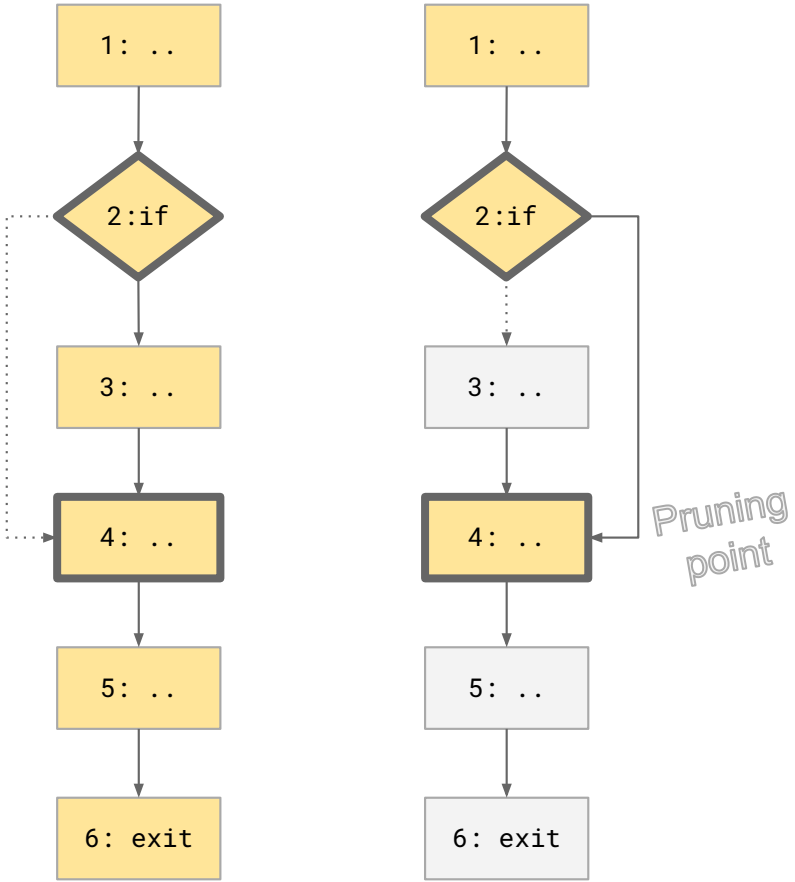
Arriving at `exit`, the verifier checks for remaining state in the stack and restart simulation from 4, taking the other branch.



State pruning: simulate execution

We restart the simulation from instruction 4 with the popped saved state.

Since instruction 4 is a pruning point, the verifier triggers the state pruning function.

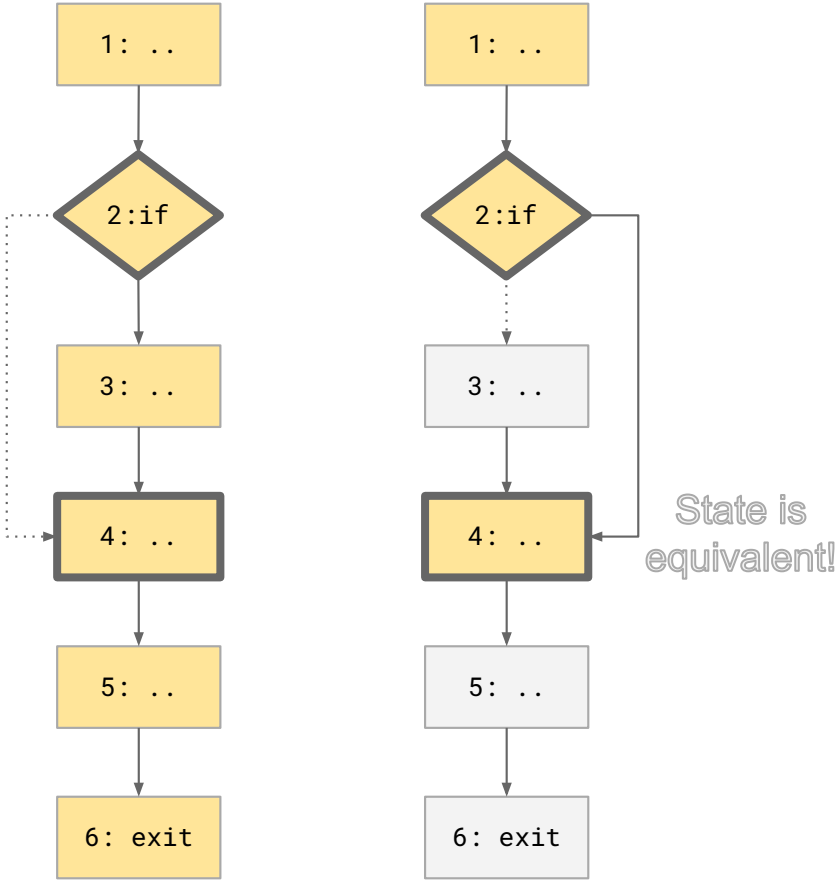


State pruning: simulate execution

We restart the simulation from instruction 4 with the popped saved state.

Since instruction 4 is a pruning point, the verifier triggers the state pruning function.

If the previous saved state is equivalent to our current state, with the skipped instruction 3, we can stop here.



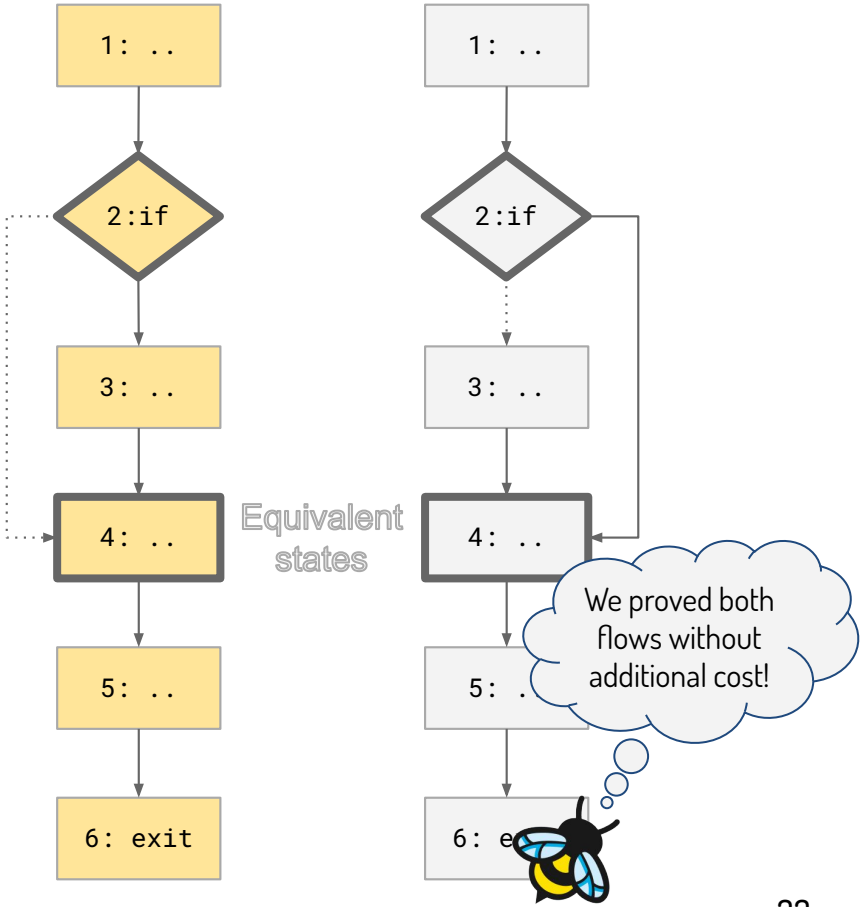
State pruning: simulate execution

We restart the simulation from instruction 4 with the popped saved state.

Since instruction 4 is a pruning point, the verifier triggers the state pruning function.

If the previous saved state is equivalent to our current state, with the skipped instruction 3, we can stop here.

The verifier simulated 6 instructions instead of initial 9.



When are states equivalent?

Current state (registers and stack) must be “included” in the saved state

- Types included
- Ranges/tnum included
- ...

current state

```
R0 = scalar(any)
R1 = scalar(0)
```

$\subseteq$

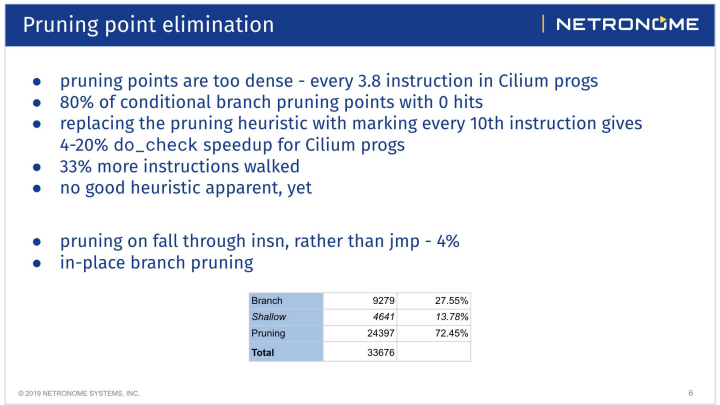
saved state

```
R0 = uninit
R1 = scalar(u64=[0; 10])
```

Heuristics to reduce the saved states

```
/* bpf progs typically have pruning point every 4 instructions [...] */
add_new_state = force_new_state;
if (env->jmps_processed - env->prev_jmps_processed >= 2 &&
    env->insn_processed - env->prev_insn_processed >= 8)
    add_new_state = true;
```

From **2589726d12a1** *("bpf: introduce bounded loops") 19 Jan 2019, v5.3.*



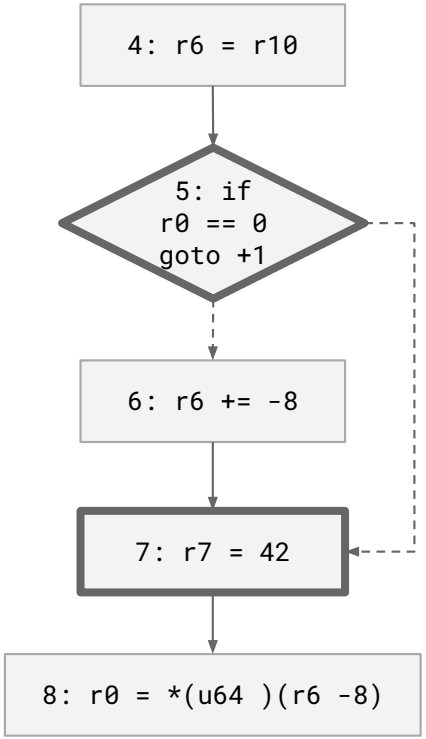
Verifier optimization work, Jakub Kicinski, May 2019 at LSF/MM/BPF

Optimizations for State Pruning

In practice, states are rarely exactly “included”.

We only need to compare what matters for verification.

The less we compare, the more efficient state pruning is.



Optimizations for State Pruning

In practice, states are rarely exactly “included”.

We only need to compare what matters for verification.

The less we compare, the more efficient state pruning is.

```
4: r6 = r10
/* Just to have two paths the
 * verifier needs to check: */
5: if r0 == 0 goto +1
6: r6 += -8
/* Pruning point: */
7: r7 = 42
8: r0 = *(u64 *)(r6 -8)
```

Register Liveness: Why?

Registers may be “dead” when we compare them.

It’s “dead” at insn X if its value is never read after X.

Register Liveness: Why?

Registers may be “dead” when we compare them.

It’s “dead” at insn X if its value is never read after X.

```
4: r6 = r10
/* Just to have two paths the
 * verifier needs to check: */
5: if r0 == 0 goto +1
6: r6 += -8
/* Pruning point: */
7: r7 = 42
8: r0 = *(u64 *)(r6 -8)
```

Register Liveness: Why?

Registers may be “dead” when we compare them.

It’s “dead” at insn X if its value is never read after X.

Ex., r6’s value at instruction 6 is read at 8. Hence, r6 is “live” between instructions 4 and 8 and “dead” before 4.

```
4: r6 = r10
/* Just to have two paths the
 * verifier needs to check: */
5: if r0 == 0 goto +1
6: r6 += -8
/* Pruning point: */
7: r7 = 42
8: r0 = *(u64 *)(r6 -8)
```

Register Liveness: How?

Propagates liveness backward from the read up to a write on the same register.

```
Live regs before insn:
  4: 0......... (bf) r6 = r10
  5: 0.....6... (1d) if r0 == 0 goto pc+1
  6: ......6... (07) r6 += -8
  7: ......6... (b7) r7 = 42
  8: ......6... (79) r0 = *(u64 *)(r6 -8)
```

Register Liveness: Example

```
4: (bf) r6 = r10    ....................  ; R6_w=fp0 R10=fp0
5: (1d) if r6 == r0 goto pc+1              ; R0_w=scalar() R6_w=fp0
6: (07) r6 += -8    ....................  ; R6=fp-8
7: (b7) r7 = 42     ....................  ; R7_w=42
8: (79) r0 = *(u64 *)(r6 -8)    .......   ; R0_w=scalar() R6=fp-24
9: (95) exit
```

Register Liveness: Example

```
4: (bf) r6 = r10    ....................   ; R6_w=fp0 R10=fp0
5: (1d) if r6 == r0 goto pc+1              ; R0_w=scalar() R6_w=fp0
6: (07) r6 += -8    ....................   ; R6=fp-8
7: (b7) r7 = 42     ....................   ; R7_w=42
8: (79) r0 = *(u64 *)(r6 -8)    ......     ; R0_w=scalar() R6=fp-24
9: (95) exit

from 5 to 7: R0_w=scalar() R6_w=fp0 R8_w=0 R10=fp0
7: R0_w=scalar() R6_w=fp0 R8_w=0 R10=fp0
7: (b7) r7 = 42    .....................   ; R7_w=42
8: (79) r0 = *(u64 *)(r6 -8)    ......     ; R0_w=scalar() R6_w=fp-16
9: (95) exit
processed 13 insns (limit 1000000) max_states_per_insn 1
total_states 1 peak_states 1 mark_read 1
```

Register Liveness: Example

```
Live regs before insn:
  4: 0......... (bf) r6 = r10
  5: 0.....6... (1d) if r0 == 0 goto pc+1
  6: ......6... (07) r6 += -8
  7: ......6... (b7) r7 = 42
  8: ......6... (79) r0 = *(u64 *)(r6 -8)
```

Register Liveness: Example

```
Live regs before insn:
  4: 0......... (bf) r6 = r10
  5: 0.....6... (1d) if r0 == 0 goto pc+1
  6: ......6... (07) r6 += -8
  7: .......... (b7) r7 = 42
  8: .......... (bf) r6 = r10
  9: ......6... (79) r0 = *(u64 *)(r6 -8)
```

Register Liveness: Example

```
Live regs before insn:
  4: 0......... (bf) r6 = r10
  5: 0.....6... (1d) if r0 == 0 goto pc+1
  6: ......6... (07) r6 += -8
  7: ......!... (b7) r7 = 42
  8: ......!... (bf) r6 = r10
  9: ......6... (79) r0 = *(u64 *)(r6 -8)
```

Register Liveness: Example

```
4:  (bf) r6 = r10   ..................  ; R6_w=fp0 R10=fp0
5:  (1d) if r6 == r0 goto pc+1   ...  ; R0_w=scalar() R6_w=fp0
6:  (07) r6 += -8   ..................  ; R6=fp-8
7:  (b7) r7 = 42    ..................  ; R7_w=42
8:  (bf) r6 = r10   ..................  ; R6_w=fp0 R10=fp0
9:  (79) r0 = *(u64 *)(r6 -8)   .....  ; R0_w=scalar() R6_w=fp0
10: (95) exit

from 5 to 7: safe
processed 12 insns (limit 1000000) max_states_per_insn 0
total_states 1 peak_states 1 mark_read 0
```

Stack Slot Liveness

Similar to register liveness:

- Computes stack slots *read* and *written* by each instruction.
- Propagates backward, until stable state is reached.

Stack Slot Liveness

Similar to register liveness:

- Computes stack slots *read* and *written* by each instruction.
- Propagates backward, until stable state is reached.

But implementation completely different.

- Range analysis is needed to know which stack slots are accessed
- Hence, *read* and *written* accumulated during do_check().

Precise Tracking: Why?

The verifier doesn't always care about the value of scalars.

We want to only compare scalar values if they are used for verification.

Ex., r0's value only matters if the program type restricts the return codes.

```
/* Just to have two paths the
 * verifier needs to check: */
5: if r0 == 0 goto +1
6: r0 = 1
/* Pruning point: */
7: r7 = 42
8: r0 = 0
9: exit
```

Precise Tracking: How?

When scalar value used for verification, precise mark is added to register and backtracked to same register (and its dependencies) in previous states.

Precise Tracking: How?

When scalar value used for verification, precise mark is added to register and backtracked to same register (and its dependencies) in previous states.

- When exiting and checking the returned value:
  - Marks r0.
- When calling a helper that takes a size argument:
  - Marks the register holding the size argument.
- Etc.

Precise Tracking: How?

When scalar value used for verification, precise mark is added to register and backtracked to same register (and its dependencies) in previous states.

1. Maintain bit array of precise registers and stack slots.
2. Backtrack through program using jump history.
3. Uses propagation rules for each instruction type:
    - dst += src => precise mark propagated to src.
    - dst = imm => precise mark cleared
    - Etc.

Precise Tracking: Example

```
5: (15) if r0 == 0x0 goto pc+1    ....   ; R0_w=scalar(umin=1)
6: (b7) r0 = 1    ....................   ; R0=1
7: (b7) r7 = 42   ....................   ; R7_w=42
8: (b7) r0 = 0    ....................   ; R0_w=0
9: (95) exit
mark_precise: frame0: last_idx 9 first_idx 7 subseq_idx -1
```

Precise Tracking: Example

```
5: (15) if r0 == 0x0 goto pc+1    ....   ; R0_w=scalar(umin=1)
6: (b7) r0 = 1    ....................   ; R0=1
7: (b7) r7 = 42   ....................   ; R7_w=42
8: (b7) r0 = 0    ....................   ; R0_w=0
9: (95) exit
mark_precise: frame0: last_idx 9 first_idx 7 subseq_idx -1
mark_precise: frame0: regs=r0 stack= before 8: (b7) r0 = 0
```

Precise Tracking: Example

```
5: (15) if r0 == 0x0 goto pc+1   ....   ; R0_w=scalar(umin=1)
6: (b7) r0 = 1   ....................   ; R0=1
7: (b7) r7 = 42   ...................   ; R7_w=42
8: (b7) r0 = 0   ....................   ; R0_w=0
9: (95) exit
mark_precise: frame0: last_idx 9 first_idx 7 subseq_idx -1
mark_precise: frame0: regs=r0 stack= before 8: (b7) r0 = 0

from 5 to 7: safe
processed 11 insns (limit 1000000) max_states_per_insn 0
total_states 1 peak_states 1 mark_read 0
```

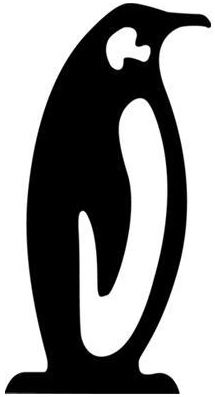
Conclusion

State pruning is a verifier optimization to walk less paths.

Main way to scale the verifier: evolved a lot over time.

More to come:

- Blog posts from the walkthrough, examples, and documents we wrote.
- Patches for logging and documentation.
- Benchmarks for pruning points.



東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025