Beyond 1M BPF instructions

Alexei Starovoitov

Meta

1 million

- It was just a number that felt big in 2019
- Now programs hit this limit often
- Main reasons:
    - always_inline
    - #pragma unroll

1 million

- It was just a number that felt big in 2019
- Now programs hit this limit often
- Main reasons:
    - always_inline
    - #pragma unroll
- Do:
    - remove always_inline. subprograms supported since 2017
    - remove #pragma. bounded loops supported since 2019
    - may or may not help with 1M limit, but surely faster run-time

Help verifier and avoid 1 million

- Use global function where possible
- Use iterators and may_goto/cond_break
- Use arena

Pros and Cons of static functions

- static function verification is path sensitive
  - each static function callsite is unique
  - for (i = 0; i < 100; i++) stat_func(i); // static function is verified up to 100 times
  - no restriction on argument types

Use global functions

- global functions are verified once, arguments are inferred from types
  - int arg1; // unknown scalar
  - int *arg1; // valid pointer or NULL
  - struct bpf_dynptr *arg2;
  - struct foo *arg3; // pointer to prog defined struct that only contains scalars or NULL
  - struct sk_buff *arg4;

Help verifier understand global functions

- global functions are verified once, arguments are inferred from types
  - int arg1; // unknown scalar
  - int *arg1; // valid pointer or NULL
  - int *arg1 __arg_nonnull; // valid pointer
  - struct bpf_dynptr *arg2;
  - struct foo *arg3; // pointer to prog defined struct that only contains scalars or NULL
  - struct sk_buff *arg4;
  - void *arg4 __arg_ctx;
  - struct task_struct *arg5 __arg_trusted __arg_nullable; // valid pointer to task, but could be NULL
  - htab_t *htab __arg_arena; // pointer to arena

Pros and Cons of bounded loops

```
for (int i = 0; i < 100; i++)  // just works, but O(n=100)

for (int i = 0; i < 10000; i++) // hits 1M insns
```

Use iterators and may_goto

- Instead of

  ```
  for (int i = 0; i < 100; i++)
  ```

- Do

  ```
  bpf_for (i, 0, 100)                                  // open coded iterator
  for (i = zero; i < 100 && can_loop; i++)     // may_goto aka can_loop aka cond_break
  ```

Bounded loops vs iterators

- Bounded loops
  - The verifier analyzes every iteration looking for termination condition within a program
  - Same state in two iterations -> reject as infinite loop
- Iterators
  - Unknown loop count, runtime will terminate it
  - may_goto allows looping 8M times or 1/4 sec
  - Same state in two iterations -> great, can prune verification early

- BPF program code needs to be written aiming opposite meaning from verifier pov

bpf_for internals

```
#define bpf_for(i, start, end) for (                                            \
        /* initialize and define destructor */                                  \
        struct bpf_iter_num ___it __attribute__((aligned(8), /* enforce, just in case */ \
                                                cleanup(bpf_iter_num_destroy))),  \
        /* ___p pointer is necessary to call bpf_iter_num_new() *once* to init ___it */ \
                            *___p __attribute__((unused)) = (                   \
                                bpf_iter_num_new(&___it, (start), (end)),       \
        /* this is a workaround for Clang bug: it currently doesn't emit BTF */ \
        /* for bpf_iter_num_destroy() when used from cleanup() attribute */     \
                                (void)bpf_iter_num_destroy, (void *)0);         \
        ({                                                                      \
                /* iteration step */                                            \
                int *___t = bpf_iter_num_next(&___it);                          \
                /* termination and bounds check */                              \
                (___t && ((i) = *___t, (i) >= (start) && (i) < (end)));         \
        });                                                                     \
)
```

may_goto internals

```
#define can_loop  \
        ({ __label__ l_break, l_continue;               \
        bool ret = true;                                \
        asm volatile goto("may_goto %l[l_break]"        \
                      :::: l_break);                    \
        goto l_continue;                                \
        l_break: ret = false;                           \
        l_continue:;                                    \
        ret;                                            \
        })

#define cond_break    \
        ({ __label__ l_break, l_continue;               \
        asm volatile goto("may_goto %l[l_break]"        \
                      :::: l_break);                    \
        goto l_continue;                                \
        l_break: break;                                 \
        l_continue:;                                    \
        })
```

Iterator pitfalls

```
bpf_for (i, 0, 100)  { arr[i]; }                // ok, O(1) insns processed

i = 0; bpf_repeat(100) { arr[i]; i++; }     // invalid access to map value

i = 0; bpf_repeat(100) { if (i < 100) arr[i]; i++; }     // hits 1M insns

i = 0; bpf_repeat(100) { if (i < 100) arr[i]; else break; i++; }     // ok, but O(n=100) insns processed

int zero = 0; // global variable
i = zero; bpf_repeat(100) { if (i < 100) arr[i]; else break; i++; }     // ok, O(1) insns processed
```

Iterator/may_goto pitfalls

```
for (i = 0; i < 100 && can_loop; i++)  { arr[i]; }       // ok, but O(n=100) insns processed

for (i = zero; i < 100 && can_loop; i++)  { arr[i]; }       // may be ok or invalid access to map value

for (i = zero; i < 100 && can_loop; i++)  { barrier_var(i); arr[i]; }     // ok, O(1) insns processed
```

Use arena

- int arr[100];
+ int __arena arr[100];

```
for (i = 0; i < 100 && can_loop; i++)  { arr[i]; }       // ok, but O(n=100) insns processed

for (i = zero; i < 100 && can_loop; i++)  { arr[i]; }   // ok, O(1) insns processed
```

Precise scalars vs wide scalars

- Two heuristics fight each other
- Several attempts last year to consolidate them
  - v5: https://lore.kernel.org/bpf/20240606005425.38285-2-alexei.starovoitov@gmail.com/

    " ... When the verifier sees 'r1 > 1000' inside the loop and it can predict it
    instead of marking r1 as precise it widens both branches, so r1 becomes
    [0, 1000] in fallthrough and [1001, UMAX] in other_branch. ..."
- Abandoned, since i += N loops cannot be handled
- Next step: scalar evolution

Scalar precision and register liveness

- Works well in bounded loops and normal code
- Cannot be trusted in iterators, since the rest of the program is not verifier yet
  - precision and liveness marks are incomplete
  - propagate_liveness() works for normal code
    - partially works (same as partially broken) within iterators
  - loop detection was fixed countless times, yet it is still somewhat broken

```
force_exact = loop_entry && loop_entry->branches > 0;
if (states_equal(env, &sl->state, cur,
                 force_exact ? RANGE_WITHIN : NOT_EXACT)) {
```

// NOT_EXACT - ok to use precision and liveness marks

Long term fix?

- Compute liveness (use - def chain) per instruction instead of built-in into verifier state chain
    - compute_live_registers() works for registers, not for stack yet
    - Big improvement for sched-ext programs
- Get rid of stack analysis
    - Convert spill/fill into extra registers R11, R12, ..., R74
    - Either in verifier or llvm (big ISA change)
    - in kernel register allocation from 74 registers into 12 on x86 or 32 on arm64
- Get rid of precision
    - Introduce data flow analysis
- Get rid of loop_entry
    - Make it per instruction instead of per state chain

BPF mission

or why we're still passionate about this code

- To innovate
  - ...
- To enable others to innovate
  - ..
- To challenge what's possible
  - When everyone says "It's impossible"

    we reply "The whole thing maybe impossible, but this part is doable".