kfuncs inlining in BPF programs

Preface

This presentation is based on the RFC [1] shared on November 2024 and some unpublished follow up work.

The RFC manages to get some performance gains for a simplistic benchmark by inlining a call to `bpf_dynptr_slice` and removing a few jumps for which conditions are proven to be constant.

Unfortunately, the RFC is a one hack on top of another and it would be interesting to hear thoughts from community.

[1] https://lore.kernel.org/bpf/20241107175040.1659341-1-eddyz87@gmail.com/

Motivation

Simplistic example

```
int dynptr_slice(struct __sk_buff *skb)
{        ...
         bpf_dynptr_from_skb(skb, 0, &psrc);
         for (i = 0; i < N; ++i) // unrolled
                 bpf_dynptr_slice(&psrc, i, NULL, 1);
         __sync_add_and_fetch(&hits, N);
         ...
}
```

Inlining a call to `bpf_dynptr_slice` and removing two conditional jumps in its body gives it a 1.53x speedup.

Motivation

Which jumps are removed in example?

```
void *bpf_dynptr_slice(const struct bpf_dynptr *p, u32 offset,
                       void *buffer__opt, u32 buffer__szk)
{
    ...
    type = bpf_dynptr_get_type(ptr);
    switch (type)
    ...
    case BPF_DYNPTR_TYPE_SKB:
            if (buffer__opt)
                    return skb_header_pointer(...);
            else
                    return skb_pointer_if_linear(...);
    ...
}
```

`bpf_dynptr_slice` approximately 40 lines of code with 10 conditionals, not very convenient to hard-code the inlined variant.

The following slides explore parts of the automated solution:

- compilation of kfuncs to BPF
- embedding of BPF code into vmlinux
- inlining mechanics
- isolated kfunc bodies verification (as in RFC)
- non-isolated kfunc bodies verification
  (as suggested by Alexei Starovoitov)

Compilation of kfuncs to BPF

Reusing kernel headers

```
#include <linux/bpf.h>
#include <linux/skbuff.h>
...
__bpf_kfunc
void *bpf_dynptr_slice(const struct bpf_dynptr *p, u32 offset,
                       void *buffer__opt, u32 buffer__szk)
{
    const struct bpf_dynptr_kern *ptr = (struct bpf_dynptr_kern *)p;
    ...
    if (buffer__opt)
        return skb_header_pointer(ptr->data, ...);
    else
        return skb_pointer_if_linear(ptr->data, ...);
    ...
}
```

Compilation of kfuncs to BPF

Makefile integration

kfuncs selected for inlining are moved to `kernel/bpf/inlinable_kfuncs.c`.

```
$(obj)/inlinable_kfuncs.bpf.bc.o: $(src)/inlinable_kfuncs.c
        $(Q)$(CC) $(c_flags) -emit-llvm -c $< -o $@

$(obj)/inlinable_kfuncs.bpf.o: $(obj)/inlinable_kfuncs.bpf.bc.o
        $(Q) $(LLC) -mcpu=v3 --mtriple=bpf --filetype=obj $< -o $@

$(obj)/inlinable_kfuncs.bpf.linked.o: $(obj)/inlinable_kfuncs.bpf.o
        $(Q)$(STRIP) --strip-debug --remove-section=.*BTF* -o $@ $<

$(obj)/verifier.o: $(obj)/inlinable_kfuncs.bpf.linked.o
```

Compilation of kfuncs to BPF

`clang -emit-llvm` + `llc` drawbacks

This keeps kernel-side and bpf-side data declarations in sync, but has a few downsides:

- it relies on compiler performing dead code elimination to remove any functions/code fragments that introduce inline assembly, as such assembly would be targeting native architecture, not BPF;
- `skb_header_pointer` and `skb_pointer_if_linear` are static inline functions defined in `skbuff.h`, whether or not inline assembly is used in these functions is outside of BPF sub-system control;
- this is `clang/llvm` specific solution, not applicable to `gcc`.

Compilation of kfuncs to BPF

Alternatives

A pipeline (during kernel build):

- `vmlinux` → `vmlinux.h` generation;
- some new `__internal_kfunc` annotation to expose non-kfunc kernel functions in the `vmlinux.h` to be used only by inlinable kfuncs;
- `inlinable_kfuncs.c` compilation;
- embedding of `inlinable_kfuncs.bpf.o` as an additional section in `vmlinux`.

But this looses static inline functions like `skb_header_pointer` and macro definitions.

Embedding of BPF code into vmlinux

BPF ELF file as data section

- use `.incbin` assembly directive to include BPF ELF object as a blob in data section;
- use ELF symbol table to find kfunc bodies in this blob;
- resolve relocations inside kfunc bodies.

```
llvm-readelf-19 --symbols kernel/bpf/inlinable_kfuncs.bpf.linked.o

Symbol table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis       Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT     2 .text
     // kfuncs with bodies
     2: 0000000000000000    40 FUNC    GLOBAL DEFAULT     2 bpf_dynptr_is_null
     3: 0000000000000030    48 FUNC    GLOBAL DEFAULT     2 bpf_dynptr_is_rdonly
     4: 0000000000000060    48 FUNC    GLOBAL DEFAULT     2 bpf_dynptr_size
     5: 0000000000000090   568 FUNC    GLOBAL DEFAULT     2 bpf_dynptr_slice
     // symbols needing relocation
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT   UND bpf_xdp_pointer
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT   UND bpf_xdp_copy_buf
     8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT   UND skb_copy_bits
```

Embedding of BPF code into vmlinux

Relocations refer to internal kernel functions

```
llvm-objdump -dr inlinable_kfuncs.bpf.linked.o
...
0000000000000090 <bpf_dynptr_slice>:
      ...
      43:    call -0x1
             0000000000000158:  R_BPF_64_32  bpf_xdp_pointer
      ...
      57:    call -0x1
             00000000000001c8:  R_BPF_64_32  bpf_xdp_copy_buf
      ...
      84:    call -0x1
             00000000000002a0:  R_BPF_64_32  skb_copy_bits
```

Lookup these functions in kernel BTF using `btf_find_by_name_kind` and patch `imm` fields of the relocated function call instructions.

Inlining mechanics

`do_check()` is modified to replace calls to kfuncs found in the blob with their bodies:

- replacement happens after main verification pass;
- if kfunc uses callee saved registers r6-r9 the spill/fill pairs are generated for these register before/after inlined kfunc body at call site;
- if kfunc uses r10 as a base pointer for load or store instructions, offsets of these instructions are adjusted;
- if kfunc uses r10 in other instructions, such r10 is considered as escaping and kfunc is not inlined.

Isolated kfunc bodies verification

- Before main verification pass, make a copy of an inlinable kfunc for each call location, the goal is to do separate dead code elimination for each kfunc call.
- When kfunc call is verified do a `push_stack()` to visit the corresponding dedicated kfunc body, put it in isolated context:
  - establish an independent call stack;
  - copy "distilled" view of parameters from the callsite;
- Proceed with usual kfunc verification logic.

Isolated kfunc bodies verification

Independent call stack

Isolate BPF program from kfunc verification, but still get some analysis result for kfunc body:

- setup frame #0 with fake representation for dynptr: two stack slots with register spills holding specific values;
- setup frame #1 with actual kfunc parameters.

The dynptr stack slot is setup in accordance to `bpf_dynptr_kern` definition:

```
struct bpf_dynptr_kern {
        void *data; // KERNEL_VALUE
        u32 size;   // size, dynptr type, read-only bit,
                    // encoded in bpf_reg_state->var_off
        u32 offset; // unknown;
```

Isolated kfunc bodies verification

Distilled view of parameters

Again, isolate BPF program state from kfunc body state, but try to preserve information useful to prune branches:

- scalars copied as-is;
- null pointers copied as-is;
- dynptr parameters are represented as pointers to stack frame #0;
- everything else is copied as `KERNEL_VALUE`.

`KERNEL_VALUE` is an opaque value to represent values originating from kernel. Any operation on `KERNEL_VALUE` returns a `KERNEL_VALUE`.

Non-isolated kfunc bodies verification

Idea

Make it possible for kfunc calls to see results from verification of prior kfunc calls. For example `bpf_dynptr_slice` can observe the effects of:

```
__bpf_kfunc int bpf_dynptr_from_skb(struct __sk_buff *s, u64 flags,
                                    struct bpf_dynptr *ptr__uninit)
{
    ...
    ptr->size |= BPF_DYNPTR_TYPE_SKB << DYNPTR_TYPE_SHIFT;
    ...
}
```

To infer the dynptr type. Then there would be no need to have special logic for fake frame and dynptr setup.

Non-isolated kfunc bodies verification

Dual representation of stack objects

For this to work verifier needs to maintain two views for a single stack object:

- one logical, used for program verification;
- one "physical", used to pass data between inlinable kfunc calls.

```
struct bpf_stack_state {
  struct bpf_reg_state spilled_ptr;
  u8 slot_type[BPF_REG_SIZE];
};
```

```
struct bpf_stack_state {
  struct bpf_reg_state spilled_ptr;
  u8 slot_type[BPF_REG_SIZE];
  /* moved from bpf_reg_state */
  u32 ref_obj_id;
  enum bpf_stack_obj_type type;
  union {
    struct { ... } dynptr;
    struct { ... } iter;
  };
};
```

Non-isolated kfunc bodies verification

Verification

- When inlinable kfunc call is verified:
  - do semantic checks as for regular kfunc;
  - distill kfunc parameters:
    - scalars and stack object pointers are passed as is;
    - everything else is passed as `KERNEL_VALUE`;
  - enter inlined function body as regular kfunc.
- Inside inlined function body refer to `bpf_stack_state->{spilled_ptr,slot_type}` when operating on stack objects.
- Upon `exit` from inlined function body return to caller, as for regular subprogram call.

Discussion

Isolated vs non-isolated kfunc bodies verification

Isolated:

- Pros:
  - no effect on the program verification logic;
  - possible to isolate in log;
  - easier to isolate effects on 1M instructions verification budget.
- Cons:
  - Require special logic to setup the fake callee frame for each stack object kind (currently only three: dynptrs, iterators and irq flags);
  - 7 call frames max, 512 bytes per each frame.

Discussion

Isolated vs non-isolated kfunc bodies verification (continued)

Non-isolated:

- Pros:
  - more generic, if verifier would model stack effects with high enough accuracy (it only tracks 8 byte aligned spills).
- Cons:
  - dual representation for stack objects adds complexity;
  - conditional instructions in the bodies of inlinable kfuncs:
    - make program verification log harder to reason;
    - potentially hit 1M instruction limit faster;
  - max 7 call frames or less, 512 bytes per each frame.

Discussion

- Which functions to inline?
  Currently the following functions are considered:
  - dynptr related functions;
  - some iterator related functions, like `num_iter_*`;
  - consider if `__htab_map_lookup_elem` can be inlined using similar mechanics.

  What else might be interesting?
- Alternatives to `clang -emit-llvm` + `llc`?