Classic BPF vs eBPF

eBPF is designed to be JITed with one to one mapping, which can also open up the possibility for GCC/LLVM compilers to generate optimized eBPF code through an eBPF backend that performs almost as fast as natively compiled code.

Some core changes of the eBPF format from classic BPF:

- Number of registers increase from 2 to 10:

  The old format had two registers A and X, and a hidden frame pointer. The new layout extends this to be 10 internal registers and a read-only frame pointer. Since 64-bit CPUs are passing arguments to functions via registers the number of args from eBPF program to in-kernel function is restricted to 5 and one register is used to accept return value from an in-kernel function. Natively, x86_64 passes first 6 arguments in registers, aarch64/ sparcv9/mips64 have 7 - 8 registers for arguments; x86_64 has 6 callee saved registers, and aarch64/sparcv9/mips64 have 11 or more callee saved registers.

  Thus, all eBPF registers map one to one to HW registers on x86_64, aarch64, etc, and eBPF calling convention maps directly to ABIs used by the kernel on 64-bit architectures.

  On 32-bit architectures JIT may map programs that use only 32-bit arithmetic and may let more complex programs to be interpreted.

  R0 - R5 are scratch registers and eBPF program needs spill/fill them if necessary across calls. Note that there is only one eBPF program (== one eBPF main routine) and it cannot call other eBPF functions, it can only call predefined in-kernel functions, though.

- Register width increases from 32-bit to 64-bit:

  Still, the semantics of the original 32-bit ALU operations are preserved via 32-bit subregisters. All eBPF registers are 64-bit with 32-bit lower subregisters that zero-extend into 64-bit if they are being written to. That behavior maps directly to x86_64 and arm64 subregister definition, but makes other JITs more difficult.

  32-bit architectures run 64-bit eBPF programs via interpreter. Their JITs may convert BPF programs that only use 32-bit subregisters into native instruction set and let the rest being interpreted.

  Operation is 64-bit, because on 64-bit architectures, pointers are also 64-bit wide, and we want to pass 64-bit values in/out of kernel functions, so 32-bit eBPF registers would otherwise require to define register-pair ABI, thus, there won't be able to use a direct eBPF register to HW register mapping and JIT would need to do combine/split/move operations for every register in and out of the function, which is complex, bug prone and slow. Another reason is the use of atomic 64-bit counters.

- Conditional jt/jf targets replaced with jt/fall-through:

  While the original design has constructs such as `if (cond) jump_true; else jump_false;`, they are being replaced into alternative constructs like `if (cond) jump_true; /* else fall-through */`.

- Introduces bpf_call insn and register passing convention for zero overhead calls from/to other kernel functions:

  Before an in-kernel function call, the eBPF program needs to place function arguments into R1 to R5 registers to satisfy calling convention, then the interpreter will take them from registers and pass to in-kernel function. If R1 - R5 registers are mapped to CPU registers that are used for argument passing on given architecture, the JIT compiler doesn't need to emit extra moves. Function arguments will be in the correct registers and BPF_CALL instruction will be JITed as single 'call' HW instruction. This calling convention was picked to cover common call situations without performance penalty.

  After an in-kernel function call, R1 - R5 are reset to unreadable and R0 has a return value of the function. Since R6 - R9 are callee saved, their state is preserved across the call.

  For example, consider three C functions:

  ```
  u64 f1() { return (*_f2)(1); }
  u64 f2(u64 a) { return f3(a + 1, a); }
  u64 f3(u64 a, u64 b) { return a - b; }
  ```

  GCC can compile f1, f3 into x86_64:

  ```
  f1:
      movl $1, %edi
      movq _f2(%rip), %rax
      jmp  *%rax
  f3:
      movq %rdi, %rax
      subq %rsi, %rax
      ret
  ```

  Function f2 in eBPF may look like:

  ```
  f2:
      bpf_mov R2, R1
      bpf_add R1, 1
  ```

```
    bpf_call f3
    bpf_exit
```

If f2 is JITed and the pointer stored to `_f2`. The calls f1 -> f2 -> f3 and returns will be seamless. Without JIT, __bpf_prog_run() interpreter needs to be used to call into f2.

For practical reasons all eBPF programs have only one argument 'ctx' which is already placed into R1 (e.g. on __bpf_prog_run() startup) and the programs can call kernel functions with up to 5 arguments. Calls with 6 or more arguments are currently not supported, but these restrictions can be lifted if necessary in the future.

On 64-bit architectures all register map to HW registers one to one. For example, x86_64 JIT compiler can map them as ...

```
R0 - rax
R1 - rdi
R2 - rsi
R3 - rdx
R4 - rcx
R5 - r8
R6 - rbx
R7 - r13
R8 - r14
R9 - r15
R10 - rbp
```

... since x86_64 ABI mandates rdi, rsi, rdx, rcx, r8, r9 for argument passing and rbx, r12 - r15 are callee saved.

Then the following eBPF pseudo-program:

```
bpf_mov R6, R1 /* save ctx */
bpf_mov R2, 2
bpf_mov R3, 3
bpf_mov R4, 4
bpf_mov R5, 5
bpf_call foo
bpf_mov R7, R0 /* save foo() return value */
bpf_mov R1, R6 /* restore ctx for next call */
bpf_mov R2, 6
bpf_mov R3, 7
bpf_mov R4, 8
bpf_mov R5, 9
bpf_call bar
bpf_add R0, R7
bpf_exit
```

After JIT to x86_64 may look like:

```
push %rbp
mov %rsp,%rbp
sub $0x228,%rsp
mov %rbx,-0x228(%rbp)
mov %r13,-0x220(%rbp)
mov %rdi,%rbx
mov $0x2,%esi
mov $0x3,%edx
mov $0x4,%ecx
mov $0x5,%r8d
callq foo
mov %rax,%r13
mov %rbx,%rdi
mov $0x6,%esi
mov $0x7,%edx
mov $0x8,%ecx
mov $0x9,%r8d
callq bar
add %r13,%rax
mov -0x228(%rbp),%rbx
mov -0x220(%rbp),%r13
leaveq
retq
```

Which is in this example equivalent in C to:

```
u64 bpf_filter(u64 ctx)
{
    return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
}
```

In-kernel functions foo() and bar() with prototype: u64 (*)(u64 arg1, u64 arg2, u64 arg3, u64 arg4, u64 arg5); will receive arguments in proper registers and place their return value into `%rax` which is R0 in eBPF. Prologue and epilogue are emitted by JIT and are implicit in the interpreter. R0-R5 are scratch registers, so eBPF program needs to preserve them across the calls as defined by calling convention.

For example the following program is invalid:

```
bpf_mov R1, 1
bpf_call foo
bpf_mov R0, R1
bpf_exit
```

After the call the registers R1-R5 contain junk values and cannot be read. An in-kernel verifier.rst is used to validate eBPF programs.

Also in the new design, eBPF is limited to 4096 insns, which means that any program will terminate quickly and will only call a fixed number of kernel functions. Original BPF and eBPF are two operand instructions, which helps to do one-to-one mapping between eBPF insn and x86 insn during JIT.

The input context pointer for invoking the interpreter function is generic, its content is defined by a specific use case. For seccomp register R1 points to seccomp_data, for converted BPF filters R1 points to a skb.

A program, that is translated internally consists of the following elements:

```
op:16, jt:8, jf:8, k:32    ==>    op:8, dst_reg:4, src_reg:4, off:16, imm:32
```

So far 87 eBPF instructions were implemented. 8-bit 'op' opcode field has room for new instructions. Some of them may use 16/24/32 byte encoding. New instructions must be multiple of 8 bytes to preserve backward compatibility.

eBPF is a general purpose RISC instruction set. Not every register and every instruction are used during translation from original BPF to eBPF. For example, socket filters are not using `exclusive add` instruction, but tracing filters may do to maintain counters of events, for example. Register R9 is not used by socket filters either, but more complex filters may be running out of registers and would have to resort to spill/fill to stack.

eBPF can be used as a generic assembler for last step performance optimizations, socket filters and seccomp are using it as assembler. Tracing filters may use it as assembler to generate code from kernel. In kernel usage may not be bounded by security considerations, since generated eBPF code may be optimizing internal code path and not being exposed to the user space. Safety of eBPF can come from the verifier.rst. In such use cases as described, it may be used as safe instruction set.

Just like the original BPF, eBPF runs within a controlled environment, is deterministic and the kernel can easily prove that. The safety of the program can be determined in two steps: first step does depth-first-search to disallow loops and other CFG validation; second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

opcode encoding

eBPF is reusing most of the opcode encoding from classic to simplify conversion of classic BPF to eBPF.

For arithmetic and jump instructions the 8-bit 'code' field is divided into three parts:

```
+----------------+--------+--------------------+
|   4 bits       |  1 bit |   3 bits           |
| operation code | source | instruction class  |
+----------------+--------+--------------------+
(MSB)                                      (LSB)
```

Three LSB bits store instruction class which is one of:

Classic BPF classes	eBPF classes
BPF_LD 0x00	BPF_LD 0x00
BPF_LDX 0x01	BPF_LDX 0x01
BPF_ST 0x02	BPF_ST 0x02
BPF_STX 0x03	BPF_STX 0x03
BPF_ALU 0x04	BPF_ALU 0x04
BPF_JMP 0x05	BPF_JMP 0x05
BPF_RET 0x06	BPF_JMP32 0x06
BPF_MISC 0x07	BPF_ALU64 0x07

The 4th bit encodes the source operand ...

```
BPF_K     0x00
BPF_X     0x08
```

- in classic BPF, this means:

```
BPF_SRC(code) == BPF_X - use register X as source operand
BPF_SRC(code) == BPF_K - use 32-bit immediate as source operand
```

- in eBPF, this means:

```
          BPF_SRC(code) == BPF_X - use 'src_reg' register as source operand
          BPF_SRC(code) == BPF_K - use 32-bit immediate as source operand
```

... and four MSB bits store operation code.

If BPF_CLASS(code) == BPF_ALU or BPF_ALU64 [in eBPF], BPF_OP(code) is one of:

```
BPF_ADD   0x00
BPF_SUB   0x10
BPF_MUL   0x20
BPF_DIV   0x30
BPF_OR    0x40
BPF_AND   0x50
BPF_LSH   0x60
BPF_RSH   0x70
BPF_NEG   0x80
BPF_MOD   0x90
BPF_XOR   0xa0
BPF_MOV   0xb0  /* eBPF only: mov reg to reg */
BPF_ARSH  0xc0  /* eBPF only: sign extending shift right */
BPF_END   0xd0  /* eBPF only: endianness conversion */
```

If BPF_CLASS(code) == BPF_JMP or BPF_JMP32 [in eBPF], BPF_OP(code) is one of:

```
BPF_JA    0x00  /* BPF_JMP only */
BPF_JEQ   0x10
BPF_JGT   0x20
BPF_JGE   0x30
BPF_JSET  0x40
BPF_JNE   0x50  /* eBPF only: jump != */
BPF_JSGT  0x60  /* eBPF only: signed '>' */
BPF_JSGE  0x70  /* eBPF only: signed '>=' */
BPF_CALL  0x80  /* eBPF BPF_JMP only: function call */
BPF_EXIT  0x90  /* eBPF BPF_JMP only: function return */
BPF_JLT   0xa0  /* eBPF only: unsigned '<' */
BPF_JLE   0xb0  /* eBPF only: unsigned '<=' */
BPF_JSLT  0xc0  /* eBPF only: signed '<' */
BPF_JSLE  0xd0  /* eBPF only: signed '<=' */
```

So BPF_ADD | BPF_X | BPF_ALU means 32-bit addition in both classic BPF and eBPF. There are only two registers in classic BPF, so it means A += X. In eBPF it means dst_reg = (u32) dst_reg + (u32) src_reg; similarly, BPF_XOR | BPF_K | BPF_ALU means A ^= imm32 in classic BPF and analogous src_reg = (u32) src_reg ^ (u32) imm32 in eBPF.

Classic BPF is using BPF_MISC class to represent A = X and X = A moves. eBPF is using BPF_MOV | BPF_X | BPF_ALU code instead. Since there are no BPF_MISC operations in eBPF, the class 7 is used as BPF_ALU64 to mean exactly the same operations as BPF_ALU, but with 64-bit wide operands instead. So BPF_ADD | BPF_X | BPF_ALU64 means 64-bit addition, i.e.: dst_reg = dst_reg + src_reg

Classic BPF wastes the whole BPF_RET class to represent a single `ret` operation. Classic BPF_RET | BPF_K means copy imm32 into return register and perform function exit. eBPF is modeled to match CPU, so BPF_JMP | BPF_EXIT in eBPF means function exit only. The eBPF program needs to store return value into register R0 before doing a BPF_EXIT. Class 6 in eBPF is used as BPF_JMP32 to mean exactly the same operations as BPF_JMP, but with 32-bit wide operands for the comparisons instead.

For load and store instructions the 8-bit 'code' field is divided as:

```
+--------+--------+-------------------+
| 3 bits | 2 bits |   3 bits          |
|  mode  |  size  | instruction class |
+--------+--------+-------------------+
(MSB)                             (LSB)
```

Size modifier is one of ...

```
BPF_W   0x00    /* word */
BPF_H   0x08    /* half word */
BPF_B   0x10    /* byte */
BPF_DW  0x18    /* eBPF only, double word */
```

... which encodes size of load/store operation:

```
B  - 1 byte
H  - 2 byte
W  - 4 byte
DW - 8 byte (eBPF only)
```

Mode modifier is one of:

```
BPF_IMM     0x00  /* used for 32-bit mov in classic BPF and 64-bit in eBPF */
BPF_ABS     0x20
BPF_IND     0x40
BPF_MEM     0x60
```

```
BPF_LEN     0x80  /* classic BPF only, reserved in eBPF */
BPF_MSH     0xa0  /* classic BPF only, reserved in eBPF */
BPF_ATOMIC  0xc0  /* eBPF only, atomic operations */
```