BPF Design Q&A

BPF extensibility and applicability to networking, tracing, security in the linux kernel and several user space implementations of BPF virtual machine led to a number of misunderstanding on what BPF actually is. This short QA is an attempt to address that and outline a direction of where BPF is heading long term.

Questions and Answers

Q: Is BPF a generic instruction set similar to x64 and arm64?

A: NO.

Q: Is BPF a generic virtual machine ?

A: NO.

BPF is generic instruction set *with* C calling convention.

Q: Why C calling convention was chosen?

A: Because BPF programs are designed to run in the linux kernel which is written in C, hence BPF defines instruction set compatible with two most used architectures x64 and arm64 (and takes into consideration important quirks of other architectures) and defines calling convention that is compatible with C calling convention of the linux kernel on those architectures.

Q: Can multiple return values be supported in the future?

A: NO. BPF allows only register R0 to be used as return value.

Q: Can more than 5 function arguments be supported in the future?

A: NO. BPF calling convention only allows registers R1-R5 to be used as arguments. BPF is not a standalone instruction set. (unlike x64 ISA that allows msft, cdecl and other conventions)

Q: Can BPF programs access instruction pointer or return address?

A: NO.

Q: Can BPF programs access stack pointer ?

A: NO.

Only frame pointer (register R10) is accessible. From compiler point of view it's necessary to have stack pointer. For example, LLVM defines register R11 as stack pointer in its BPF backend, but it makes sure that generated code never uses it.

Q: Does C-calling convention diminishes possible use cases?

A: YES.

BPF design forces addition of major functionality in the form of kernel helper functions and kernel objects like BPF maps with seamless interoperability between them. It lets kernel call into BPF programs and programs call kernel helpers with zero overhead, as all of them were native C code. That is particularly the case for JITed BPF programs that are indistinguishable from native kernel C code.

Q: Does it mean that 'innovative' extensions to BPF code are disallowed?

A: Soft yes.

At least for now, until BPF core has support for bpf-to-bpf calls, indirect calls, loops, global variables, jump tables, read-only sections, and all other normal constructs that C code can produce.

Q: Can loops be supported in a safe way?

A: It's not clear yet.

BPF developers are trying to find a way to support bounded loops.

Q: What are the verifier limits?

A: The only limit known to the user space is BPF_MAXINSNS (4096). It's the maximum number of instructions that the unprivileged bpf program can have. The verifier has various internal limits. Like the maximum number of instructions that can be explored during program analysis. Currently, that limit is set to 1 million. Which essentially means that the largest program can consist of 1 million NOP instructions. There is a limit to the maximum number of subsequent branches, a limit to the number of nested bpf-to-bpf calls, a limit to the number of the verifier states per instruction, a limit to the number of maps used by the program. All these limits can be hit with a sufficiently complex program. There are also non-numerical limits that can cause the program to be rejected. The verifier used to recognize only pointer + constant expressions. Now it can recognize pointer + bounded_register. bpf_lookup_map_elem(key) had a requirement that 'key' must be a pointer to the stack. Now, 'key' can be a pointer to map value. The verifier is steadily getting 'smarter'. The limits are being removed. The only way to know that the program is going to be accepted by the verifier is to try to load it. The bpf development process guarantees that the future kernel versions will accept all bpf programs that were accepted by the earlier versions.

Instruction level questions

Q: LD_ABS and LD_IND instructions vs C code

Q: How come LD_ABS and LD_IND instruction are present in BPF whereas C code cannot express them and has to use builtin intrinsics?

A: This is artifact of compatibility with classic BPF. Modern networking code in BPF performs better without them. See 'direct packet access'.

Q: BPF instructions mapping not one-to-one to native CPU

Q: It seems not all BPF instructions are one-to-one to native CPU. For example why BPF_JNE and other compare and jumps are not cpu-like?

A: This was necessary to avoid introducing flags into ISA which are impossible to make generic and efficient across CPU architectures.

Q: Why BPF_DIV instruction doesn't map to x64 div?

A: Because if we picked one-to-one relationship to x64 it would have made it more complicated to support on arm64 and other archs. Also it needs div-by-zero runtime check.

Q: Why there is no BPF_SDIV for signed divide operation?

A: Because it would be rarely used. llvm errors in such case and prints a suggestion to use unsigned divide instead.

Q: Why BPF has implicit prologue and epilogue?

A: Because architectures like sparc have register windows and in general there are enough subtle differences between architectures, so naive store return address into stack won't work. Another reason is BPF has to be safe from division by zero (and legacy exception path of LD_ABS insn). Those instructions need to invoke epilogue and return implicitly.

Q: Why BPF_JLT and BPF_JLE instructions were not introduced in the beginning?

A: Because classic BPF didn't have them and BPF authors felt that compiler workaround would be acceptable. Turned out that programs lose performance due to lack of these compare instructions and they were added. These two instructions is a perfect example what kind of new BPF instructions are acceptable and can be added in the future. These two already had equivalent instructions in native CPUs. New instructions that don't have one-to-one mapping to HW instructions will not be accepted.

Q: BPF 32-bit subregister requirements

Q: BPF 32-bit subregisters have a requirement to zero upper 32-bits of BPF registers which makes BPF inefficient virtual machine for 32-bit CPU architectures and 32-bit HW accelerators. Can true 32-bit registers be added to BPF in the future?

A: NO.

But some optimizations on zero-ing the upper 32 bits for BPF registers are available, and can be leveraged to improve the performance of JITed BPF programs for 32-bit architectures.

Starting with version 7, LLVM is able to generate instructions that operate on 32-bit subregisters, provided the option -mattr=+alu32 is passed for compiling a program. Furthermore, the verifier can now mark the instructions for which zero-ing the upper bits of the destination register is required, and insert an explicit zero-extension (zext) instruction (a mov32 variant). This means that for architectures without zext hardware support, the JIT back-ends do not need to clear the upper bits for subregisters written by alu32 instructions or narrow loads. Instead, the back-ends simply need to support code generation for that mov32 variant, and to overwrite bpf_jit_needs_zext() to make it return "true" (in order to enable zext insertion in the verifier).

Note that it is possible for a JIT back-end to have partial hardware support for zext. In that case, if verifier zext insertion is enabled, it could lead to the insertion of unnecessary zext instructions. Such instructions could be removed by creating a simple peephole inside the JIT back-end: if one instruction has hardware support for zext and if the next instruction is an explicit zext, then the latter can be skipped when doing the code generation.

Q: Does BPF have a stable ABI?

A: YES. BPF instructions, arguments to BPF programs, set of helper functions and their arguments, recognized return codes are all part of ABI. However there is one specific exception to tracing programs which are using helpers like bpf_probe_read() to walk kernel internal data structures and compile with kernel internal headers. Both of these kernel internals are subject to change and can break with newer kernels such that the program needs to be adapted accordingly.

Q: Are tracepoints part of the stable ABI?

A: NO. Tracepoints are tied to internal implementation details hence they are subject to change and can break with newer kernels. BPF programs need to change accordingly when this happens.

Q: How much stack space a BPF program uses?

A: Currently all program types are limited to 512 bytes of stack space, but the verifier computes the actual amount of stack used and both interpreter and most JITed code consume necessary amount.

Q: Can BPF be offloaded to HW?

A: YES. BPF HW offload is supported by NFP driver.

Q: Does classic BPF interpreter still exist?

A: NO. Classic BPF programs are converted into extend BPF instructions.

Q: Can BPF call arbitrary kernel functions?

A: NO. BPF programs can only call a set of helper functions which is defined for every program type.

Q: Can BPF overwrite arbitrary kernel memory?

A: NO.

Tracing bpf programs can *read* arbitrary memory with bpf_probe_read() and bpf_probe_read_str() helpers. Networking programs cannot read arbitrary memory, since they don't have access to these helpers. Programs can never read or write arbitrary memory directly.

Q: Can BPF overwrite arbitrary user memory?

A: Sort-of.

Tracing BPF programs can overwrite the user memory of the current task with bpf_probe_write_user(). Every time such program is loaded the kernel will print warning message, so this helper is only useful for experiments and prototypes. Tracing BPF programs are root only.

Q: New functionality via kernel modules?

Q: Can BPF functionality such as new program or map types, new helpers, etc be added out of kernel module code?

A: NO.

Q: Directly calling kernel function is an ABI?

Q: Some kernel functions (e.g. tcp_slow_start) can be called by BPF programs. Do these kernel functions become an ABI?

A: NO.

The kernel function protos will change and the bpf programs will be rejected by the verifier. Also, for example, some of the bpf-callable kernel functions have already been used by other kernel tcp cc (congestion-control) implementations. If any of these kernel functions has changed, both the in-tree and out-of-tree kernel tcp cc implementations have to be changed. The same goes for the bpf programs and they have to be adjusted accordingly.